Distributed services across the network from edge to core

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (30^{th}cycle)

Distributed services across the network from edge to core

By

Amedeo Sapio

******

Supervisor(s):
Prof. Mario Baldi

Doctoral Examination Committee:
Prof. Mario Nemirovsky, Referee, Barcelona Supercomputing Center
Dr. Domenico Siracusa, Referee, Create-Net
Prof. Enzo Mingozzi, Università di Pisa
Prof. Lorenzo De Carli, Colorado State University
Prof. Fulvio Risso, Politecnico di Torino

Politecnico di Torino
2018

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Amedeo Sapio
2018

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

To my family

Abstract

The current internet architecture is evolving from a simple carrier of bits to a platform able to provide multiple complex services running across the entire Network Service Provider (NSP) infrastructure. This calls for increased flexibility in resource management and allocation to provide dedicated, on-demand network services, leveraging a distributed infrastructure consisting of heterogeneous devices. More specifically, NSPs rely on a plethora of low-cost Customer Premise Equipment (CPE), as well as more powerful appliances at the edge of the network and in dedicated data-centers.

Currently a great research effort is spent to provide this flexibility through Fog computing, Network Functions Virtualization (NFV), and data plane programmability. Fog computing or Edge computing extends the compute and storage capabilities to the edge of the network, closer to the rapidly growing number of connected devices and applications that consume cloud services and generate massive amounts of data. A complementary technology is NFV, a network architecture concept targeting the execution of software Network Functions (NFs) in isolated Virtual Machines (VMs), potentially sharing a pool of general-purpose hosts, rather than running on dedicated hardware (i.e., appliances). Such a solution enables virtual network appliances (i.e., VMs executing network functions) to be provisioned, allocated a different amount of resources, and possibly moved across data centers in little time, which is key in ensuring that the network can keep up with the flexibility in the provisioning and deployment of virtual hosts in today's virtualized data centers. Moreover, recent advances in networking hardware have introduced new programmable network devices that can efficiently execute complex operations at line rate. As a result, NFs can be (partially or entirely) folded into the network, speeding up the execution of distributed services.

The work described in this Ph.D. thesis aims at showing how various network services can be deployed throughout the NSP infrastructure, accommodating to the

different hardware capabilities of various appliances, by applying and extending the above-mentioned solutions. First, we consider a data center environment and the deployment of (virtualized) NFs. In this scenario, we introduce a novel methodology for the modelization of different NFs aimed at estimating their performance on different execution platforms. Moreover, we propose to extend the traditional NFV deployment outside of the data center to leverage the entire NSP infrastructure. This can be achieved by integrating native NFs, commonly available in low-cost CPEs, with an existing NFV framework. This facilitates the provision of services that require NFs close to the end user (e.g., IPsec terminator). On the other hand, resource-hungry virtualized NFs are run in the NSP data center, where they can take advantage of the superior computing and storage capabilities.

As an application, we also present a novel technique to deploy a distributed service, specifically a web filter, to leverage both the low latency of a CPE and the computational power of a data center. We then show that also the core network, today dedicated solely to packet routing, can be exploited to provide useful services. In particular, we propose a novel method to provide distributed network services in core network devices by means of task distribution and a seamless coordination among the peers involved. The aim is to transform existing network nodes (e.g., routers, switches, access points) into a highly distributed data acquisition and processing platform, which will significantly reduce the storage requirements at the Network Operations Center and the packet duplication overhead.

Finally, we propose to use new programmable network devices in data center networks to provide much needed services to distributed applications. By offloading part of the computation directly to the networking hardware, we show that it is possible to reduce both the network traffic and the overall job completion time.

Contents

List of Figures

List of Tables

Nomenclature

Acronyms / Abbreviations

ACK Acknowledgment Number

ALU Arithmetic Logic Unit

ASIC Application-Specific Integrated Circuit

BNG Broadband Network Gateway

BSP Bulk Synchronous Parallel

CAS Column Access Strobe

CGNAT Carrier-Grade Network Address Translation

CPE Customer Premise Equipment

DAIET Data Aggregation In nETwork

DATS Intel Dataplane Automated Testing System

DC Data Center

DDIO Intel Data Direct I/O Technology

DDR Double Data Rate

DPDK Intel Data Plane Development Kit

DPI Deep Packet Inspection

DPPD Intel Data Plane Performance Demonstrators

DSL Digital Subscriber Line

EO Elementary Operation

ETSI European Telecommunications Standards Institute

FCS Frame Check Sequence

GPGPU General-Purpose computing on Graphics Processing Units

GRE Generic Routing Encapsulation

IE − pair Ingress-Egress pair

IID Ingress ID

IoT Internet of Things

ISP Internet Service Provider

IXP Internet eXchange Point

LAN Local Area Network

LSI Logical Switching Instance

MEDINA Massively Distributed Network Data Caching Platform

ML Machine Learning

MPI Message Passing Interface

MPLS MultiProtocol Label Switching

MSS Maximum Segment Size

NAT Network Address Translation

NF Network Functions

NFV Network Functions Virtualization

NFVI NFV Infrastructure

NFVO NFV Orchestrator

NHLFE Next Hop Label Forwarding Entry

NIC Network Interface Card

NNF Native Network Functions

NSP Network Service Providers

OvS Open vSwitch

POP Point Of Presence

PROX Intel Packet pROcessing eXecution Engine

RMT Reconfigurable Match Tables

RTT Round Trip Time

SACK TCP Selective Acknowledgment

SDN Software Defined Networking

SEQ Sequence Number

SGD Stochastic Gradient Descent

SRAM Static Random Access Memory

SSSP Single Source Shortest Path

TC Linux Traffic Control

TCAM Ternary Content-Addressable Memory

ToR Top-of-Rack

TTL Time To Live

UI User Interface

UN Universal Node

URL Universal Resource Locator

VIM Virtual Infrastructure Manager

VLAN Virtual Local Area Network

VM Virtual Machine

VNF Virtualized Network Function

WAN Wide Area Network

WCC Weakly Connected Components

Chapter 1

Introduction

Modern telecom companies are no longer simple providers of telephone/internet connectivity. In fact, they are increasingly relying on selling value-added services to boost their revenues. Connectivity plans are often bundled with music and video streaming [1–3], safe-browsing, anti-malware and parental control [4] services that, when coupled with zero-rating [5, 6], can be especially appealing to customers.

On the other hand, the recent trend of network "softwarization" is paving the way to commoditization of telecommunications infrastructure. In fact, today many of the fixed-function middleboxes can be replaced by software network functions, potentially sharing a pool of general-purpose hosts, providing more flexibility, simpler management and configuration, and lower time to market for new functionalities. This transition is spearheaded by the NFV technology that proposes the execution of software network functions in isolated VMs rather than on dedicated hardware. By using SDN, network traffic can be steered through a chain of Virtualized Network Functions (VNFs) in order to provide aggregated services. Network Service Providers (NSPs) can leverage this flexibility to easily deploy new, on-demand, services for both their internal use (e.g., monitoring) and their customers (e.g., web cache).

Currently, NSPs rely on a heterogeneous infrastructure composed of high speed networking appliances (e.g., routers), a large amount of low-cost, resource-limited Customer Premise Equipment (CPE), as well as more powerful appliances at the edge of the network and in dedicated data-centers, as showed in Figure 1.1. In

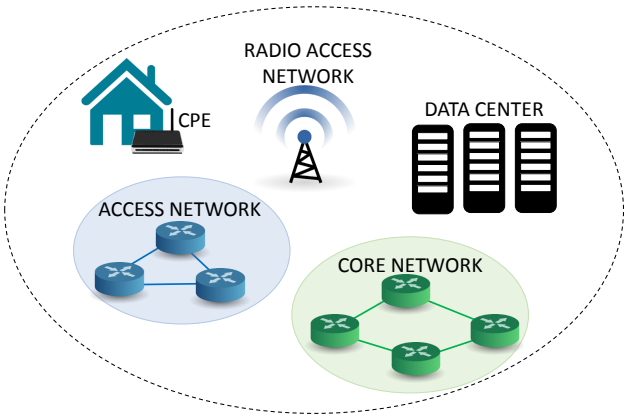


Fig. 1.1 Telecom operator infrastructure.

this dissertation, we show how all these different domains can be used to provide additional services suited to their specific constraints and limitations.

First, chapter 2 introduces a methodology for the modelization of network functions (NFs) focused on the identification of recurring execution patterns as basic building blocks and aimed at providing a platform independent representation. By mapping each modeling building block on specific hardware, the performance of the network function can be estimated in terms of maximum throughput that the network function can achieve on the specific execution platform. The approach is such that once the basic modeling building blocks have been mapped, the estimate can be computed automatically for any modeled network function. Experimental results on several sample network functions show that although our approach cannot be very accurate without taking in consideration traffic characteristics, it is very valuable for those application where even loose estimates are key. One such example is orchestration in NFV platforms, as well as in general virtualization platforms where virtual machine placement is based also on the performance of network services offered to them. Being able to automatically estimate the performance of a VNF on different execution hardware, enables optimal placement of VNFs themselves as well as the virtual hosts they serve, while efficiently utilizing available resources.

While chapter 2 describes a unified modeling approach for generic NFs, in chapter 3 we focus on software NFs running in a virtualized environment. These are often implemented using VMs because they provide an isolated environment compatible with classical cloud computing technologies. This isolation is required to leverage consolidation in a data center and comes at a substantial cost in terms

of required resources. We propose to extend the NFV infrastructure to support the deployment of services closer to the end user, using resource-constrained devices such as residential CPEs. This solution is especially beneficial for low-latency services. However, these devices cannot provide the large amount of resources required for deploying standard VMs. Nevertheless, such hardware often runs a Linux-based operating system that supports several software modules (e.g., iptables) that can be used to implement network functions (e.g., a firewall), which can be exploited to provide some of the services offered by simple VNFs, with reduced overhead. We also propose and validate an architecture that integrates those native software components in a NFV platform, making their use transparent from the user's point of view. This integration allows to jointly orchestrate simple NFs executed in the CPE with low hardware resources and complex VNFs running in the data center, hence combining the benefits of the cloud with the locality of the services running on local CPEs.

An instance of service that can be decomposed in a lightweight component running on residential gateways and a resource intensive task running in the data center is presented in detail in chapter 4. Given that nowadays users access content mostly through mobile apps and web services, both based on HTTP, several filtering applications, such as parental control, malware detection, and corporate policy enforcement, require inspecting Universal Resource Locators (URLs) contained in HTTP requests. Currently, such filtering is most commonly performed in end devices or in middleboxes. Filtering applications running on end devices are less resource intensive because they operate only on traffic from a single user and possibly leverage a hook at the HTTP level to access protocol data, but it is left to the user whether to execute them. On the other hand, middleboxes present the challenge of ensuring that they lay on the path of all the traffic from any relevant device. Residential gateways seem to be the ideal place where to implement traffic filtering because they forward all traffic generated by the hosts on home(-office) networks. However, these devices usually have very limited computation and memory resources, while URL-based filtering is quite demanding. In fact existing approaches rely on a large database of rules coupled with either deep packet inspection or transparent proxying for URL extraction. In chapter 4 we present *U-Filter*, a URL filtering solution based on a distributed architecture where a lightweight, efficient URL extraction and policy enforcement component runs on residential gateways, delegating to a remote policy server the resource intensive task of verifying policy compliance.

Thanks to the lightweight communication between the two components and the very limited resource requirements of the local module, U-Filter (*i*) can be deployed on resource-limited devices such as residential gateways, and (*ii*) has almost no impact on the performance of the device, as well as on the users' browsing experience, as demonstrated by the presented experiments.

Traffic analysis and monitoring is of paramount importance also in the access and core network devices. While the main limitation of residential gateways is their limited hardware capability, in these devices the main challenge is given by the massive amount of data that they must forward at high speed. The analysis of this large traffic is key to many domains including network management, security, network forensics. Traditionally, it is performed by a NF, running on a dedicated device, accessing traffic at a specific point within the network through a link tap or a port of a node mirroring packets. This approach is problematic because the dedicated device must be equipped with a large amount of computation and storage resources to cope with the task. Alternatively, in order to achieve scalability, analysis can be performed by multiple NF instances running in a cluster of hosts. However this is normally located at a remote location with respect to the observation point, hence requiring to move across the network a large volume of captured traffic. To address this problem we present an algorithm to distribute the task of capturing, processing and storing packets traversing a network across multiple packet forwarding nodes (e.g., IP routers).

Essentially, our solution, presented in chapter 5, allows individual nodes on the path of a flow to operate on subsets of packets of that flow in a completely distributed and decentralized manner. The solution is based on having each node traversed by a traffic flow act on a subset of the packets of the flow. The proposed algorithm allows nodes to independently agree on which one is capturing which packets in a way that ensures that each packet is captured by at least n nodes, where n is a parameter of the algorithm, which can be set to 1 to minimize overhead or to a higher value to achieve redundancy. Nodes temporarily store the fraction of packets allocated to them and create a distributed index that enables efficient retrieval of packets (e.g., for forensics applications). With minimal changes to the implementation, the basic principles of the presented solution can be applied to the distributed execution of generic tasks on data flowing through a network of nodes with processing and storage capabilities. This has applications in various fields ranging from Fog Computing, to microservice architectures, to the Internet of Things.

Finally, we present an initial proposal to leverage the high speed data center network fabric to provide services to speed up distributed applications. The advent of flexible networking hardware and expressive data plane programming languages have produced networks that are deeply programmable. By delegating part of the computation to the networking hardware, distributed data center applications can not only reduce the computation done by the CPU, but also reduce the traffic at each network device, effectively decreasing job completion times. However, it is not clear yet what kinds of computation should be delegated to the network. In chapter 6 we discuss the opportunities and challenges for co-designing data center distributed systems with their network layer. We argue that in-network computation tasks must be judiciously crafted to match the limitations of the network machine architecture of programmable devices. With the help of our experiments on machine learning and graph analytics workloads, we identify that aggregation functions raise opportunities to exploit the limited computation power of networking hardware to lessen network congestion and improve the overall application performance. Moreover, as a proof-of-concept, we propose DAIET, a system that performs in-network data aggregation.

In fact, many scalable data center applications follow a partition-aggregate pattern where data and computations are distributed among many servers and their partial results are exchanged over the network and aggregated to produce the final output. For these workloads, the network communication costs are often one of the dominant scalability bottlenecks. Experimental results with an initial prototype show a large data reduction ratio (86.9%-89.3%) and a similar decrease in the workers' computation time without requiring severe application-level modifications.

Modern networks are becoming smarter through a combination of new hardware and a higher level of programmability. This thesis aims at showing how the entire network infrastructure, from the home network to the data center network, can provide additional services tailored to the specific capabilities and requirements of different domains.

Chapter 2

Network Function Modeling and Performance Estimation

2.1 Introduction

For a few years now software network appliances have been increasingly deployed. Initially, their appeal stemmed from their lower cost, shorter time-to-market, ease of upgrade when compared to purposely designed hardware devices. These features are particularly advantageous in the case of appliances, a.k.a. middleboxes, operating on relatively recent, higher layer protocols that are usually more complex and are possibly still evolving. More recently, with the overwhelming success and diffusion of cloud computing and virtualization, software appliances became natural means to ensure that network functionalities have the same flexibility and mobility as the virtual machines (VMs) they offer services to. In this context, implementing in software even less complex, more stable network functionalities is valuable. This trend led to embracing *Software Defined Networking* (SDN) and *Network Functions Virtualization* (NFV). The former as a hybrid hardware/software approach to ensure high performance for lower layer packet forwarding, while retaining a high degree of flexibility and programmability. The latter as a virtualization solution targeting the execution of software network functions in isolated VMs sharing a pool of hosts, rather than on dedicated hardware (i.e., appliances). Such a solution enables virtual network appliances (i.e., VMs executing network functions) to be provisioned,

The content of this chapter has been described in [7–9].

allocated a different amount of resources, and possibly moved across data centers in little time, which is key in ensuring that the network can keep up with the flexibility in the provisioning and deployment of virtual hosts in today's virtualized data centers. Additional flexibility is offered when coupling NFV with SDN as network traffic can be steered through a chain of *Virtualized Network Functions* (VNFs) in order to provide aggregated services. With inputs from the industry, the NFV approach has been standardized by the European Telecommunications Standards Institute (ETSI) in 2013 [10].

The flexibility provided by NFV requires the ability to effectively assign compute nodes to VNFs and allocate the most appropriate amount of resources, such as CPU quota, RAM, virtual interfaces. In the ETSI standard the component in charge of taking such decisions is called *orchestrator* and it can also dynamically modify the amount of resources assigned to a running VNF when needed. The orchestrator can also request the migration of a VNF when the current compute node executing it is no longer capable of fulfilling the VNF performance requirements. These tasks require the orchestrator to be able to estimate the performance of VNFs according to the amount of resources they can use. Such estimation must take into account the nature of the traffic manipulation performed by the VNF at hand, some specifics of its implementation, and the expected amount of traffic it operates on. A good estimation is key in ensuring higher resource usage efficiency and avoid adjustments at runtime.

This chapter proposes a unified modeling approach applicable to any VNF, independently of the platform it is running on. By mapping a VNF model on a specific hardware it is possible to predict the maximum amount of traffic that the VNF can sustain with the required performance. The proposed modeling approach relies on the identification of the most significant operations performed by the VNF on the most common packets. These operations are described in a hardware independent notation to ensure that the model is valid for any execution platform. The mapping of the model on a target hardware architecture (required in order to determine the actual performance) can be automated, hence allowing to easily apply the approach to each available hardware platform and choose the most suitable for the execution.

Even if the proposed modeling approach has been defined with NFV in mind, it can be applied to non-virtualized network functions (NFs), whether implemented in

software or hardware, provided that the implementation and characteristics of the underlying hardware are known. The availability of a unified modeling approach for VNF and NF is instrumental in the integration of middleboxes in an NFV infrastructure [11], which is important in a transition phase and for specific applications where a dedicated or specialized hardware platform is necessary for a specific NF to satisfy performance requirements.

The modeling approach is introduced in Section 2.2 and is illustrated in Section 2.3 by applying it to various network functions. In order to validate the proposed models, Section 2.4 compares the estimated performance with actual measurements of software network functions running on a general purpose hardware platform. After discussing related work in Section 2.5, Section 2.6 concludes the chapter.

2.2 Methodology

The proposed modeling approach is based on the definition of a set of processing steps, here called *Elementary Operations* (EOs), that are common throughout various NF implementations. This stems from the observation that, generally, most NFs perform a rather small set of operations when processing the average packet, namely, well-defined alteration of packet header fields, coupled with data structure lookups.

An EO is informally defined as the longest sequence of elementary steps (e.g., CPU instructions or ASIC transactions) that is common among the processing tasks or multiple NFs. As a consequence, an EO has variable granularity ranging from a simple I/O or memory load operation, to a whole IP checksum computation. On the other hand, EOs are defined so that each can be potentially used in multiple NF models.

An NF is modeled as a sequence of EOs that represent the actions performed for the vast majority of packets. Since we are interested in performance estimation, we ignore operations that affects only a small number of packets (i.e., less the 1%), since these tasks have a negligible impact on performance, even when they are more complex and resource intensive than the most common ones. Accordingly exceptions, such as failures, configuration changes, etc., are not considered.

It is important to highlight that NF models produced with this approach are hardware independent, which ensures that they can be applied when NFs are deployed

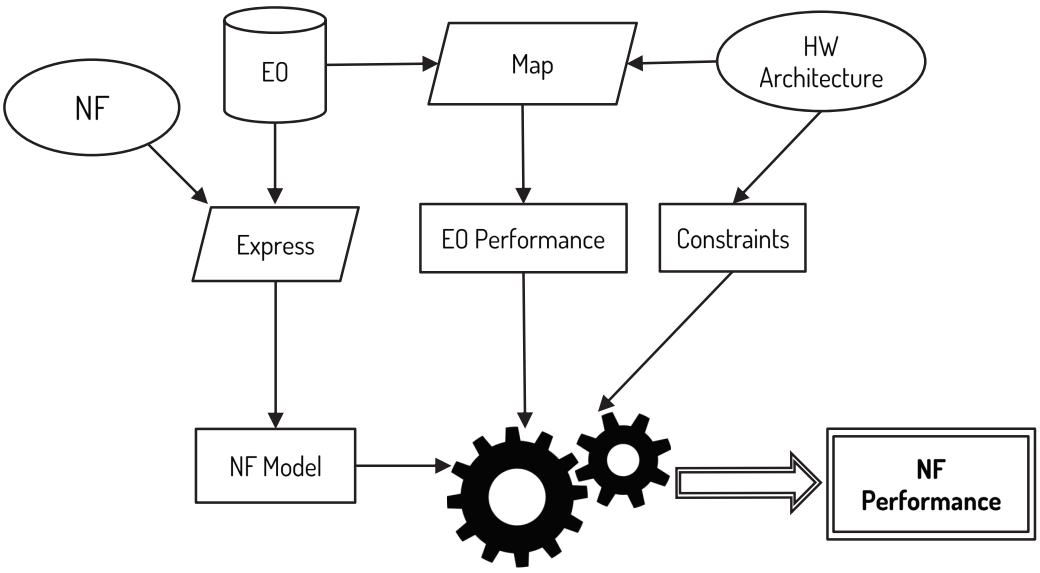


Fig. 2.1 NF modeling and performance estimation approach.

on different execution platforms. In order to estimate the performance of an NF on a specific hardware platform, each EO must be *mapped* on the hardware components involved in its execution and their features. This mapping allows to take into consideration the limits of the involved hardware components and gather a set of parameters that affect the performance (e.g., clock frequency). Moreover, the load incurred by each component when executing each EO must be estimated, whether through actual experiments or based on nominal hardware specifications. The data collected during such mapping are specific to EOs and the hardware platform, but not to a particular NF. Hence, they can be applied to estimate the performance of any NF modeled in terms of EOs. Specifically, the performance of each individual EO involved in the NF model is computed and composed considering the cumulative load that all EOs impose on the hardware components of the execution platform, while heeding all of the applicable constraints. Figure 2.1 summarizes the steps and intermediate outputs of the proposed approach.

Table 2.1 presents a list of sample EOs that we identified when modeling a number of NFs. Such list is by no means meant to be exhaustive; rather, it should be incrementally extended whenever it turns out that a new NF being considered cannot be described in terms of previously identified EOs. When defining an EO, it is important to identify the parameters related to traffic characteristics that significantly affect the execution and resource consumption.

Table 2.1 List of sample EOs

	EO	Parameters	Description
1	`I/O_mem` `mem_I/O`	hdr, data	Packet copy between I/O and (cache) memory
2	`parse` `deparse`	b	Parse or encapsulate a data field
3	`increase` `decrease`	b	Increase/decrease a field
4	`sum`	b	Sum 2 operands
5	`checksum` `inc_checksum`	b	Compute IP checksum
6	`array_access`	es, max	Direct access to a byte array in memory
7	`ht_lookup`	N, HE, max, p	Simple hash table lookup
8	`lpm_lookup`	b, es	Longest prefix match lookup
9	`ct_insertion`	N, HE, max, p	Cache table insertion

2.2.1 Elementary Operations

A succinct description of the EOs listed in table 2.1 is provided below.

1. *Packet copy between I/O and memory*:

   A packet is copied from/to an I/O buffer to/from memory or CPU cache. *hdr* is the number of bytes that are preferably stored in the fastest cache memory, while *data* bytes can be kept in lower level cache or main memory. The parameters have been chosen taking into consideration that some NPUs provide a manual cache that can be explicitly loaded with the data that need fast access. General purpose CPUs may have assembler instructions (e.g., `PREFETCHh`) to explicitly influence the cache logic.

2. *Parse or encapsulate a data field*:

   A data field of *b* bytes stored in memory is parsed. A parsing operation is necessary before performing any computation on a field (it corresponds to loading a processor register). The dual operation, i.e., *deparse*, implies storing back into memory a properly constructed sequence of fields.

3. *Increase/decrease a field*:

   Increase/decrease the numerical value contained in a field of b bytes. The field to increase/decrease must have already been parsed.

4. *Sum two operands*:

   Two operands of b bytes are added.

5. *Compute IP checksum*:

   The standard IP checksum computation is performed on b bytes. When only some bytes change in the relevant data, the checksum can be computed incrementally from the previous correct value [12]. In this case, the previous value of the checksum must be parsed beforehand and b is the number of changed bytes for which the checksum must be incrementally computed.

6. *Direct access to a byte array in memory*:

   This EO performs a direct access to an element of an array in memory using an index. Each array entry has size *es*, while the array has at most *max* entries.

7. *Simple hash table lookup*:

   A simple lookup in a direct hash table is performed. The hash key consists of N components and each entry has size equal to *HE*. The table has at most *max* entries and the collision probability is p.

8. *Longest Prefix Match lookup*:

   This EO selects an entry from a table based on the Longest Prefix Match (LPM). This lookup algorithm selects the most specific of the matching entries in a table (i.e., the one where the largest number of leading bits of the key match those in the table entry). The parameter b represents the number of bytes, on average, of the matching prefix, while *es* is the entry size.

9. *Cache table insertion*: Save in a hash table an entry with the current timestamp or update the timestamp if the entry is already present. This EO have the same parameters of the simple hash table lookup operation; the performance of both EOs depends from the hash table characteristics.

For the sake of simplicity (and without affecting the validity of the approach, as shown by the results in Section 2.4), in modeling NFs by means of EOs, we assume

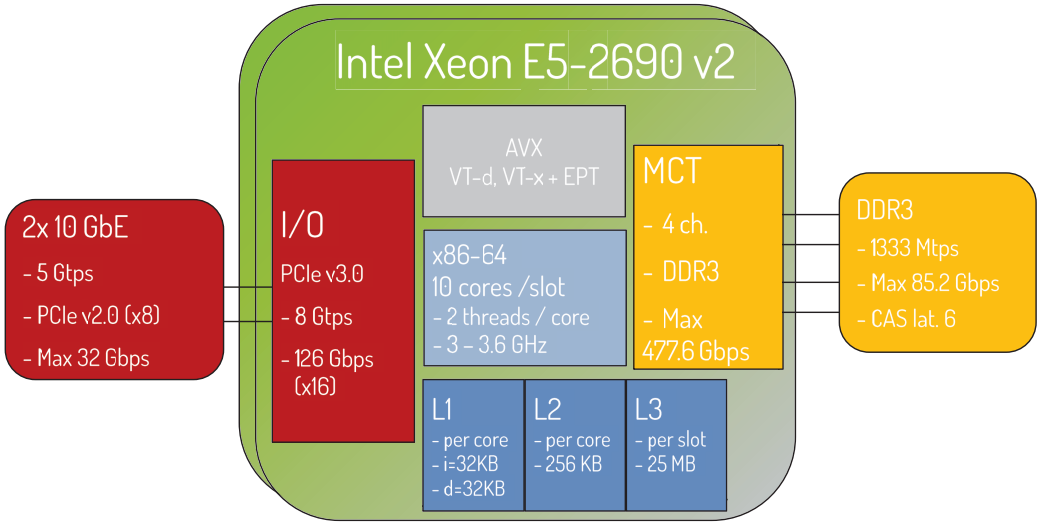


Fig. 2.2 Hardware architecture description.

that the number of processor registers is larger than the number of packet fields that must be processed simultaneously. Therefore there is no competition for processor registers.

2.2.2 Mapping to Hardware

We now proceed to map the described EOs to a specific hardware platform: a server with 2 Intel Xeon E5-2690 v2 CPUs (Ivy Bridge architecture with ten physical cores at 3 GHz), 64 GB DDR3 RAM memory and one Intel 82599ES network card with 2x10Gbps Ethernet ports. Figure 2.2 provides a schematic representation of the platform main components and relative constraints using the template proposed in [13].

Using the CPU reference manual [14], it is possible to determine the operations required for the execution of each EO in Table 2.1 and estimate the achievable performance.

1. `I/O_mem(hdr, data)` - `mem_I/O(hdr, data)`

The considered CPU provides a DMA-like mechanism to move data from the I/O buffers to the shared L3 cache and viceversa. Intel DPDK drivers [15] with Data Direct I/O Technology (DDIO) leverage this capability to move packets to/from the

L3 cache without the CPU intervention, improving the packet processing speed. The portion of each packet that must be processed (*hdr*) is then moved from L3 cache into the L1/L2 cache by the CPU. This operation requires 31 clock cycles to access the L3 cache, around 5 cycles to write a L1/L2 cache line and 9 cycles to write back a L3 cache line [16]. On the whole, the execution of this EO requires:

$$31 + [5|9] * \lceil \frac{hdr}{64B} \rceil \text{ clock cycles}$$

provided that *hdr* is less than the total amount of L1 and L2 caches, as it is reasonable for modern systems and common packet sizes. The multiplier is 5 for `I/O_mem` and 9 for `mem_I/O`.

2. `parse(b) - deparse(b)`

Loading a 64 bit register requires 5 clock cycles if data is in L1 cache or 12 clock cycles if data is in L2 cache, otherwise an additional L3 cache or DRAM memory access is required to retrieve a 64 byte line and store it in L1 or L2 respectively (the reverse operation has the same cost):

$$5 * \lceil \frac{b}{8B} \rceil \text{ clock cycles } \{+\lceil \frac{b}{64B} \rceil \text{ L3 or DRAM accesses}\}$$

or

$$12 * \lceil \frac{b}{8B} \rceil \text{ clock cycles } \{+\lceil \frac{b}{64B} \rceil \text{ L3 or DRAM accesses}\}$$

3. `increase(b) - decrease(b)`

Whether a processor includes an increase (decrease) instruction or one for adding (subtract) a constant value to a 64 bit register, this EO requires 1 clock cycle to complete. However, thanks to pipelining, up to 3 independent such instructions can be executed during 1 clock cycle:

$$\lceil 0.33 * \frac{b}{8B} \rceil \text{ clock cycles}$$

```
Register ECX: number of bytes b
Register EDX: pointer to the buffer
Register EBX: checksum

CHECKSUM_LOOP:

     XOR    EAX, EAX    ;EAX=0
     MOV    AX, WORD PTR [EDX]  ;AX <- next word
     ADD    EBX, EAX    ;add to checksum
     SUB    ECX, 2  ;update number of bytes
     ADD    EDX, 2  ;update buffer
     CMP    ECX, 1  ;check if ended
     JG CKSUM_LOOP

MOV EAX, EBX    ;EAX=EBX=checksum
;EAX=checksum>>16 EAX is the carry
SHR EAX, 16
AND EBX, 0xffff ;EBX=checksum&0xffff
;EAX=(checksum>>16)+(checksum&0xffff)
ADD EAX, EBX
MOV EBX, EAX    ;EBX=checksum
SHR EBX, 16 ;EBX=checksum>>16
ADD EAX, EBX    ;checksum+=(checksum>>16)
MOV checksum, EAX   ;checksum=EAX
```

Fig. 2.3 Sample Intel x86 assembly code for checksum computation.

4. `sum(b)`

On the considered architecture, the execution of this EO is equivalent to the EO `increase(b)`. Please note that this is not necessarily the case on every architecture.

5. `checksum(b) - inc_checksum(b)`

Figure 2.3 shows a sample assembly code to compute a checksum on an Intel x86-64 processor. Assuming that the data on which the checksum is computed is not in L1/L2 cache, according to the Intel documentation [14], the execution of this code

requires

$$7 * \lceil \frac{b}{2} \rceil + 8 \text{ clock cycles}$$
$$+ \lceil \frac{b}{64B} \rceil \text{ L3 or DRAM accesses}$$

6. `array_access(es, max)`

Direct array access needs to execute an "`ADD`" instruction (1 clock cycle) for computing the index and a "`LOAD`" instruction resulting into a direct memory access and as many clock cycles as the number of CPU registers required to load the selected array element:

$$1 + \lceil \frac{es}{8B} \rceil \text{ clock cycles } + \lceil \frac{es}{64B} \rceil \text{ DRAM accesses}$$

7. `ht_lookup(N, HE, max, p)`

We assume that a simple hash table lookup is implemented according to the pseudo-code described in [13] and shown in Figure 2.4 for ease of reference.

Considering that the hash entry needs to be loaded from memory to L1 cache, a simple hash table lookup would require approximately:

$$\lceil (4 * N + 106 + 5 * \lceil \frac{HE}{8B} \rceil + 5 * \lceil \frac{HE}{32B} \rceil) * (1 + p) \rceil$$

clock cycles and

$$\lceil (\lceil \frac{HE}{64B} \rceil * (1 + p)) \rceil \text{ L3 or DRAM accesses}$$

Otherwise, if the entry is already in the L1/L2 cache, the memory accesses and cache store operations are not required. Notice that in order for the whole table to be in cache, its size should be limited by:

$$max * HE \leq 32KB + 256KB = 288KB$$

```
Register $1-N: key components
Register $HL: hash length
Register $HP: hash array pointer
Register $HE: hash entry size
Register $Z: result

Pseudo code:
     # hash key calculation
     eor $tmp, $tmp
     for i in 1 ... N
         eor $tmp, $i
     # key is available in $tmp

     # calculate hash index from key
     udiv $tmp2, $tmp, $HL
     mls $tmp2, $tmp2, $HL, $tmp
     # index is available in $tmp2

     # index -> hash entry pointer
     mul $tmp, $tmp2, $HE
     add $tmp, $HP
     # entry pointer available in $tmp

     <prefetch entry to L1 memory>
     # pointer to L1 entry -> $tmp2

     # hash key check (entry vs. key)
     for i in 1 ... N
         ldr $Z, [$tmp2], #4
         # check keys
         cmp $i, $Z
         bne collision
     # no jump means matching keys
     # pointer to data available in $Z
```

Fig. 2.4 Hash table lookup pseudo-code.

```
 Register $HE: updated hash entry
 Register $HT: pointer to previous L1 entry
 Register $HS: hash entry size

Pseudo code:
     for i in 1 ... $HS/8
         mov [$HT], $HE
         add $HT, #8

     #update timestamp
     rdtsc
     mov [$HT], EDX
     add $HT, #2
     mov [$HT], EAX

     <store updated entry>
```

Fig. 2.5 Entry update pseudo-code for cache table insertion.

8. `lpm_lookup(b,es)`

There are several different algorithms for finding the longest matching rule. Here we consider the *DIR-24-8* algorithm [17], which in most cases (when the entry matches up to 24 bits) is able to find the first matching rule with only one memory access. This speed, however, comes at the cost of space, because of the redundant storage of rules. However, the very fast lookup this algorithm provides heavily outweighs this space constraint. With the *DIR-24-8* algorithm the longest prefix match requires the equivalent of an `array_access(es,16M)` operation if $b \leq 3$, otherwise an additional memory access is required, corresponding to an `array_access(es,255)`.

9. `ct_insertion(N, HE, max, p)`

The EO corresponds to a lookup in a hash table followed by either the insertion of a new entry or the update of the timestamp in an existing one. The two operations have approximately the same cost; the pseudo-code in Figure 2.5 shows the operations required to update the timestamp of the entry.

As a result the cache table insertion algorithm would require approximately:

$$\lceil (4*N+129+7*\lceil\frac{HE}{8B}\rceil+5*\lceil\frac{HE}{32B}\rceil)*(1+p)\rceil$$

clock cycles and

$$2*\lceil(\lceil\frac{HE}{64B}\rceil*(1+p))\rceil \text{ L3 or DRAM accesses}$$

2.3 Modeling Use Cases

This section demonstrates the application of the modeling approach described in section 2.2. EOs are used to describe the operation of simple network functions, such as L2 Switches, and a more complex case, a *Broadband Network Gateway* (BNG). The model is used to estimate the performance of each use case on the hardware platform presented in Section 2.2.2. The accuracy of the estimation is evaluated in Section 2.4 based on real measurements obtained through a range of experiments.

2.3.1 L2 Switch

First we model an Ethernet switch with a static forwarding table. In this case the output port is selected through a simple lookup in the table using the destination MAC address. Afterwards we consider a more general case where the forwarding table is populated using the backward learning algorithm. Finally, we model an MPLS switch, which selects the output interface according to the MPLS label in the packet.

Basic Forwarding

For each packet the switch selects the output interface where it must be forwarded; such interface is retrieved from a hash table using as a key the destination MAC address extracted from the packet.

More in detail, when a network interface receives a packet, it stores it in an I/O buffer. In order to access the Ethernet header, the CPU/NPU must first copy the packet in cache or main memory (possibly with the help of a Direct Memory Access

module). Since the switch operates only on the Ethernet header together with the identifier of the ingress and egress ports through which it is received and forwarded, the corresponding 30 bytes ($18+6+6$ bytes)[1] are copied in the fastest cache, while the rest of the packet (up to 1500 bytes) can be kept in L3 cache or main memory. To ensure generality, we consider that an incoming packet cannot be copied directly from an I/O buffer to another, but instead it must be first copied in (cache) memory.

The switch must then read the destination MAC address (6 bytes) prior to using it to access the hash table to get the appropriate output interface. The hash table has one key (the destination MAC) and consists of 12 byte entries composed of the key and the output interface MAC address. A common number of entries in a typical switch implementation is $\approx 2M = 2 \times 2^{20}$, which gives an idea, when mapping the model to a specific hardware, of whether the hash table can be fully stored in cache under generic traffic conditions. The new output port must be stored in the data structure in L3 cache or main memory (which, as previously explained, has the same cost as parsing 6 bytes), before moving the packet to the buffer of the selected output I/O device.

The resulting model expressing the above steps in terms of EOs is summarized in Figure 2.6a, where `ps` is the ethernet payload size. Such model assumes that the collision probability of the hash is negligible (i.e., the hash table is sufficiently sparse).

Applying to the Ethernet switch model the mapping of EOs presented in Section 2.2.2, we can estimate that forwarding a packet, regardless of the packet size (thanks to DDIO), requires:

$$213 \text{ clock cycles } + 1 \text{ DRAM access}$$

As a consequence, a single core of an Intel Xeon E5-2690v2 operating at *3.6 Ghz* can process $\approx$ *17.31 Mpps*, while the DDR3 memory can support *111.08 Mpps*.

The memory throughput is estimated considering that each packet requires a 12 byte memory access to read the hash table entry, which has a latency of:

$$\frac{(\text{CAS latency} \times 2) + 3}{\text{data rate}}$$

[1] We consider that interfaces are identified by their Ethernet address. Different implementations can use a different identifier, which leads to a minor variation in the model.

I/O_mem(30,ps)
parse(6)
ht_lookup(1,12,2M,0)
deparse(6)
mem_I/O(30,ps)

(a) Basic forwarding switch model.

I/O_mem(30,ps)
parse(8)
ht_lookup(1,14,2M,0)
parse(12)
ct_insertion(2,14,2M,0)
deparse(6)
mem_I/O(30,ps)

(b) Learning switch model.

I/O_mem(34,ps-4)
parse(3)
ht_lookup(1,12,1M,0)
parse(1)
decrease(1)
deparse(10)
mem_I/O(34,ps-4)

(c) MPLS switch model.

Fig. 2.6 Models of different L2 switches.

If we consider minimum size (64 bytes) packets (i.e., an unrealistic, worst case scenario), a single core can process $\approx$ *11.36 Gbps*.

Learning Switch

We here consider an Ethernet switch with VLAN support, in which case the key used for lookups in the forwarding table consists of the destination MAC address and the VLAN ID (2 bytes). Hence, 8 bytes must be parsed from the header (destination address and VLAN ID) of each packet in order to obtain the lookup key and entries in the forwarding table are 14 bytes long (destination address and VLAN ID as key and output interface as value). Since the switch is applying backward learning, for each packet the source MAC address and source port are used to update the forwarding table. The switch must also parse the source MAC address and read from

memory the source port (added to packets stored in memory) and either add an entry in the forwarding table or just update the timestamp of an existing one. The resulting model is shown in Figure 2.6b.

When mapped to our hardware architecture, forwarding a packet requires an estimated:

$$352 \text{ clock cycles } + 2 \text{ DRAM accesses}$$

hence the maximum throughput reachable by a single core is reduced to $\approx$ *10.47 Mpps*, while the DDR3 memory can support *55.54 Mpps*. This translates to a maximum throughput of $\approx$ *6.87 Gbps* for 64 byte packets.

MPLS Switch

An MPLS switch is a simple, yet currently widely deployed, Network Function. For each packet the switch swaps a single MPLS label and forwards the packet on an Ethernet network towards the next hop. The new label and the next hop are retrieved from a hash table whose key is the label extracted from the packet. Since the MPLS switch modifies the label in the MPLS header, in addition to associating to it the output port, the MPLS header (4 bytes) is also preferably copied in the L1/L2 cache, while the rest of the packet can be kept in L3 cache or main memory. The switch must then extract the MPLS label (20 bit $\approx$ 3 bytes) prior to using it to access the hash table to get the new label and the next hop. The hash table has one key (the label) and consists of 12 byte entries:

- Input label (key) - 3 bytes
- Output label - 3 bytes
- Next hop Ethernet address - 6 bytes.

The maximum number of entries in the hash table is, in the worst case, $1M = 2^{20}$ and we consider that the collision probability is negligible.

In the most general case, each entry, referred in the MPLS standard documents as Next Hop Label Forwarding Entry (NHLFE), could hold more than one label in case of multiple label operations. For the sake of simplicity we model only a single

label operation: the swapping of a label, which is the most frequent operation in common MPLS switch deployment scenarios.

The switch must also decrease the Time-To-Live (TTL) contained in the MPLS header, which requires parsing the corresponding field, followed by a decrease operation for the 1 byte field. The new (outgoing) MPLS header and output port must be stored in main memory (encapsulation of 10 bytes) and moved to the buffer of the output I/O device. The resulting model is summarized in Figure 2.6c.

As we map this model to the considered hardware platform, we can conclude that the estimated forwarding cost for a MPLS switch is:

$$224 \text{ clock cycles } + 1 \text{ DRAM access}$$

corresponding to a maximum per core throughput of $\approx$ *16.45 Mpps*, while the memory could provide the same throughput as the basic forwarding switch. The maximum bitrate considering 64 bytes packets is $\approx$ *10.8 Gbps*.

2.3.2 Broadband Network Gateway

A Broadband Network Gateway (BNG) is the first IP point in the network for DSL and cable modem subscribers connecting them to the broadband IP network. The primary task of a BNG is to aggregate traffic from various subscriber sessions from an access network, and route it to the core network of the service provider. Moreover, a BNG carries out additional vital tasks for Network Service Providers (NSPs), such as managing subscribers' sessions, performing accounting and enforcing operator policies. Hence, a BNG represents a more complex use case for the application of the proposed modelization approach.

In our modeling effort we refer to the software implementation of a BNG present in the Intel Data Plane Performance Demonstrators (DPPD) [18]. This is an open source, highly optimized software BNG specifically intended for performance analysis. In this implementation the traffic in the access network between the Customer Premise Equipment (CPE) and the BNG is encapsulated using Ethernet QinQ frames, while the traffic between the BNG and the Carrier-grade NAT (CGNAT) in the core MPLS network is encapsulated using GRE (Generic Routing Encapsulation). In this scenario packets received from the access network and packets received from

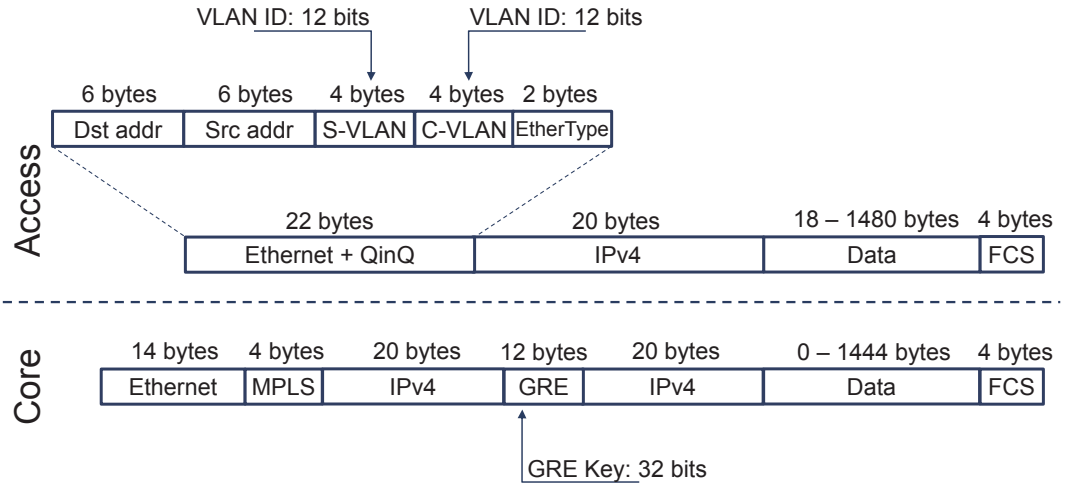


Fig. 2.7 Packet formats.

the core network are processed differently by the BNG, thus 2 separate models are required for the 2 directions. The two different formats of packets forwarded in the access and in the core network is illustrated in Figure 2.7.

Packets from CPEs are matched with 2 different tables: (*i*) a hash table that given the QinQ tag provides the corresponding GRE key (up to 16M entries of 7 bytes) and (*ii*) an LPM routing table that given the destination IP address returns the output port, the IP address of the remote GRE tunnel endpoint, the next hop MAC address and the MPLS label (this table can contain up to 8K routes). Packets from the core network are instead matched with only one hash table that given the GRE key and the inner destination IP address provides the QinQ tag, the destination MAC address and the output port. The BNG supports up to 64K CPEs, thus this table can contain up to 64K entries of 23 bytes. The QinQ tag and the GRE key are used to track the subscriber (e.g., for accounting), while the tunnel endpoint (i.e., the CGNAT) is selected according to the destination of the packet.

The resulting models for both directions are summarized in Figure 2.8. When processing packets from the access network, MAC with QinQ and IP headers are loaded preferably in L1/L2 cache, so that the QinQ header can be parsed. The extracted QinQ tag is used for the lookup in table (*i*), while the destination IP address is parsed and deployed in the LPM lookup table (*ii*). These 2 lookups provide the output GRE key, destination IP and MAC addresses, MPLS tag and output port that are used in the encapsulation of the output packet. The TTL (Time To Live) of

Packet from access network
I/O_mem(42,ps-20)
parse(8)
ht_lookup(1,7,16M,0)
parse(4)
lpm_lookup(2,23)
parse(1)
decrease(1)
parse(2)
inc_checksum(1)
checksum(ps-14)
sum(2)
checksum(20)
parse(16)
ct_insertion(2,23,64K,0)
deparse(70)
mem_I/O(70,ps-20)

Packet from core network
I/O_mem(70,ps-56)
parse(8)
ht_lookup(2,23,64K,0)
parse(1)
decrease(1)
parse(2)
inc_checksum(1)
deparse(42)
mem_I/O(42,ps-56)

Fig. 2.8 BNG model.

the internal IP packet is decremented and thus the checksum must be incrementally updated starting from the current value. The new packet format requires also the computation of the GRE checksum and the external IP packet Total Length field and header checksum. Moreover, backward learning is used to populate the table used to process packets from the core network. Hence, an additional `ct_insertion` operation is required, after parsing source port, MAC and IP addresses. The final packet is formed with the encapsulation of 70 bytes, corresponding to the new ethernet, MPLS, external IP, GRE and inner IP headers and then sent to the output I/O buffer.

Packets from the core network require a parse operation for the GRE key and the inner destination IP before using them for an hash table lookup to get the QinQ tag, the destination MAC address and the output port. In this case also the TTL of the

inner IP packet is decremented and the checksum incrementally updated. The new outgoing packet must then be stored in memory or cache (encapsulation of 42 bytes) and moved to the buffer of the output I/O device.

Mapping these models to the considered hardware platform, we can conclude that the estimated cost to process a 64 bytes packet from the access network is:

$$717 \text{ clock cycles } + 6 \text{ DRAM accesses}$$

corresponding to a maximum per core throughput of $\approx$ *5.14 Mpps* (*3.37 Gbps*), while the DDR3 memory can support $\approx$ *12.11 Mpps* (*7.95 Gbps*). The estimated cost to process a 64 byte packet from the core network is:

$$274 \text{ clock cycles } + 1 \text{ DRAM access}$$

corresponding to a maximum per core throughput $\approx$ *13.45 Mpps* (*8.83 Gbps*) and $\approx$ *24.68 Mpps* (*16.2 Gbps*) achievable by the DDR3 memory.

2.4 Experimental validation

In order to evaluate the accuracy of the estimates produced by the proposed modeling approach, in this section we present measurements made in a lab setting with software implementations of the presented Network Functions.

2.4.1 L2 Switch

As a software L2 switch we deploy an instance of Open vSwitch [19] configured through the OpenFlow protocol to select an output port based on the destination MAC address. The switch is used with both a predefined forwarding table and backward learning. Moreover, the same switch implementation is also configured to perform MPLS label swapping. The software switch runs on the hardware platform presented in Figure 2.2.

To minimize the interference of the operating system drivers, the network interfaces are managed through the Intel DPDK drivers [15]. These drivers are designed for fast packet processing, providing the possibility to receive and send packets

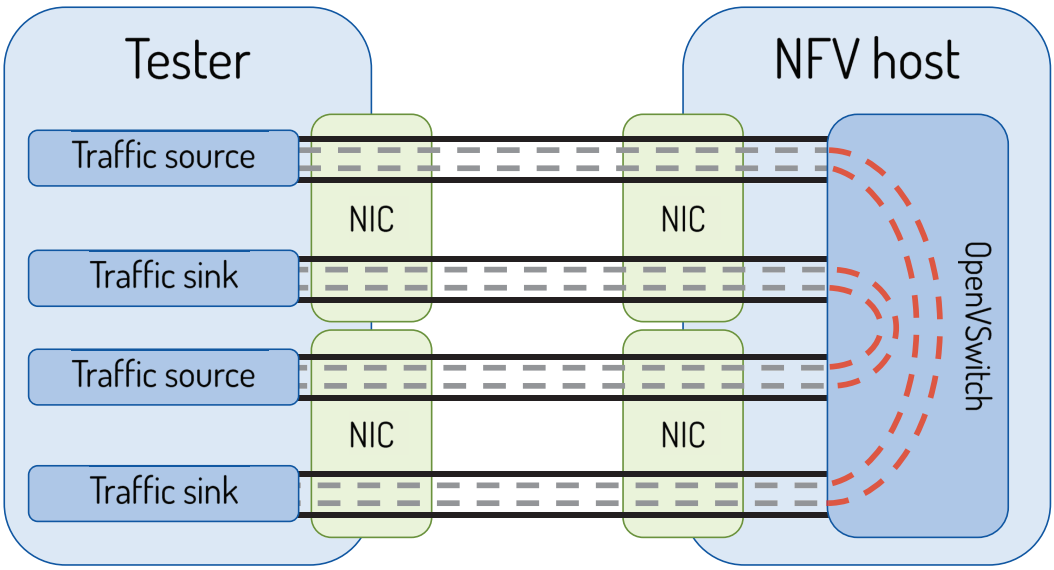


Fig. 2.9 L2 Switch testbed setup.

directly from/to a network interface card within the minimum possible number of CPU cycles. In fact, DPDK drivers allow the CPU to receive packets using polling, rather than interrupts, since interrupt service routines execute a number of additional operations for each packet. Moreover, with DPDK drivers it is possible to leverage DDIO to load packets directly in the L3 cache with no overhead for the CPU.

A separate PC with the same hardware configuration is used as a traffic generator leveraging PF_RING/DNA drivers [20] to generate traffic up to the link capacity even with packets of minimum size. As shown in Figure 2.9, we run 4 different processes, 2 PF_RING senders and 2 PF_RING counters, pinned on different dedicated cores, to generate traffic on both NICs at line rate and, at the same time, compute statistics on received packets. All the tests are run for 5 minutes and the results present the averaged aggregate statistics on both sinks.

Basic Forwarding

To test the forwarding performance of the software switch, we generate traffic consisting of Ethernet packets with ever different destination MAC addresses, in order to prevent inter-packet caching. For each destination address we had previously added a rule in the switch to set the destination port. The resulting throughput for

different packet sizes is presented in Figure 2.10, together with the values estimated with the modeling approach in Section 2.3.1.

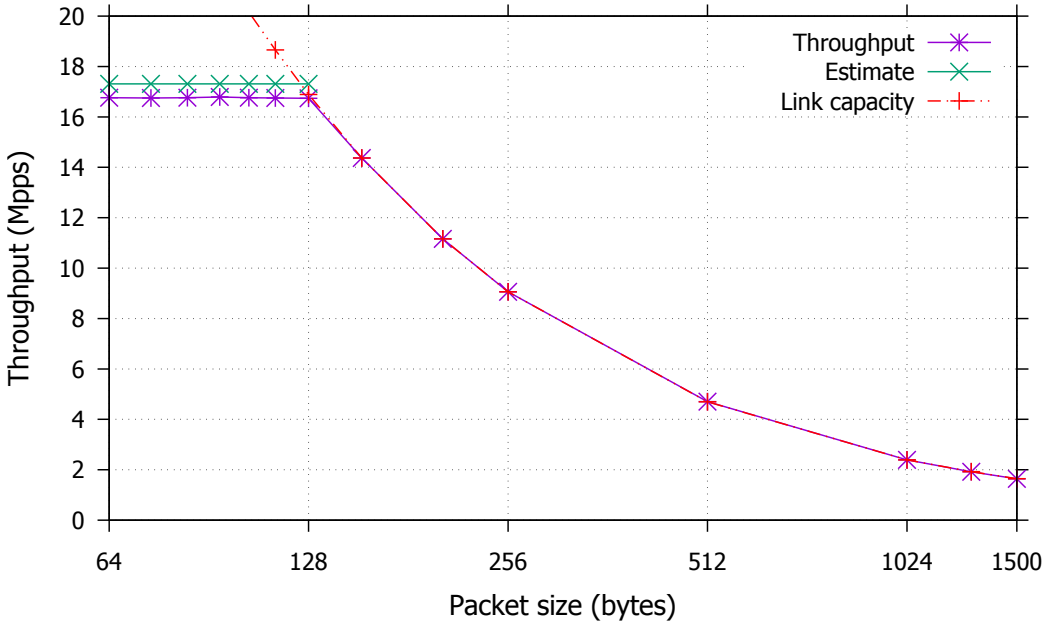


Fig. 2.10 Basic forwarding performance.

The experimental results show that in this scenario the switch can achieve throughput up to the link capacity except with packets smaller than 128 bytes. For values of the performance estimate that exceed the link capacity (i.e., packets greater than 128 bytes), our model cannot be applied by itself as it considers the hardware computational capability and not the transmission rate of the physical links that becomes the limiting factor in such scenario. With smaller packets, our mode estimates a rate around *17 Mpps*, regardless of the actual packet size. The measurements demonstrate that that the throughput estimation is quite accurate, with only a 4% error.

Learning Switch

The next test is aimed at measuring to what extent the performance of the software switch is impacted in a context in which the learning algorithm plays a significant role in the processing being performed. We configure the switch by pushing an OpenFlow rule with a "`NORMAL`" action, so that it acts as a regular layer 2 learning switch [21]. Then, before starting the test, for each destination address that will be used in the test traffic, the corresponding traffic sink sends a packet with the same

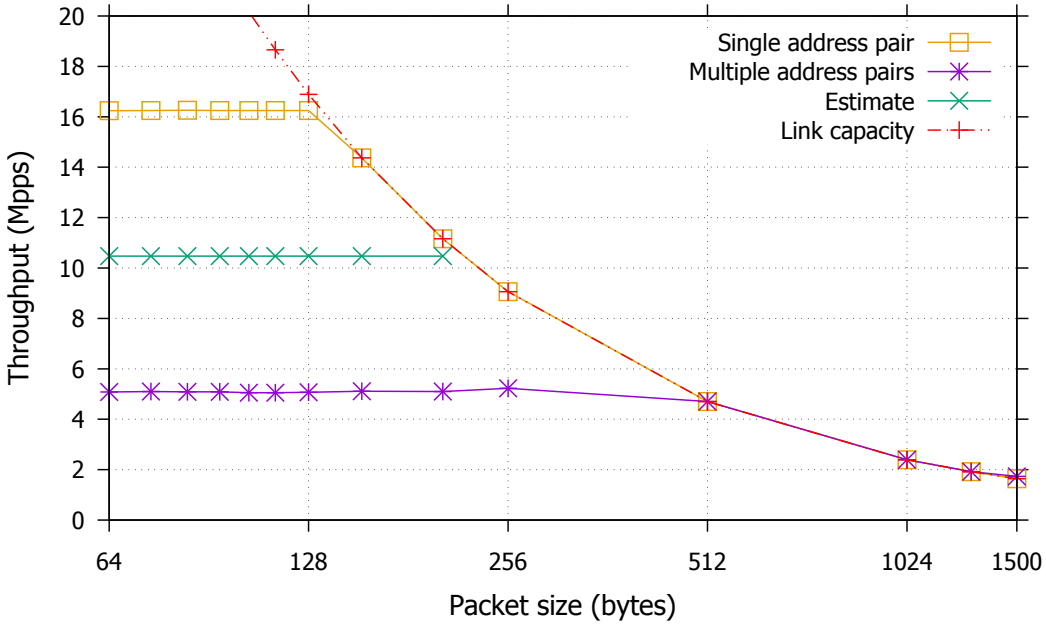


Fig. 2.11 Learning switch performance.

address as source. This allows the switch to learn the output port associated with the addresses to ensure that measurements will be taken in a steady state (i.e., avoiding that some packets are flooded on all ports, while others are sent out on a specific port).

To isolate the impact of caching on the performance we consider 2 scenarios. In a first test each traffic source sends traffic addressed to only one destination and with a unique source address. In a second test each traffic source sends traffic using repeatedly 10 different source and destination addresses. We chose this number of different address pairs after a preliminary evaluation, which showed that this is the turning point at which the throughput experiences a sharp decrease due to cache misses. The difference between the basic forwarding throughput (see Figure 2.10) and the throughput in this second scenario represents the performance degradation due to the learning functionality itself. The resulting throughput measured in both tests by the traffic sinks, is presented in Figure 2.11 for different packet sizes, together with the values estimated with the proposed model.

The results show that, when all the packets have a single source and destination address pair, the switch can achieve a very high throughput ($\approx$ *16 Mpps* with packets that are 128 bytes or smaller) because the 2 corresponding entries (one for forwarding

and one for learning) are matched within the *microflow* cache [19] that Open vSwitch implements in kernel space. Since the microflow cache is stored in L1/L2 cache (thanks to its small size), the execution of the learning code updating the timestamp requires very few clock cycles ($\approx$ *140*), significantly lower than our estimate ($\approx$ *350*) that assumes a main memory-based lookup of the forwarding table (because the Open vSwitch implementation specific microflow cache is not modeled).

On the contrary, with 10 different addresses the throughput radically drops to only *5 Mpps* for packets smaller than 512 bytes because the forwarding entry to be updated is not in the microflow cache, in which case the Open vSwitch implementation delegates the update operation to a user space process. This is significantly different from the estimated throughput of $\approx$ *10.5 Mpps* expected when an entry timestamp is updated for each packet. While the design choice of Open vSwitch performing packet processing, beyond basic forwarding, in user space increases flexibility and configurability, it adds a large overhead. As the test show, this complexity is not considered in the model. On the other hand, our model also does not capture the microflow cache-based optimization and the unlikely case in which it allows to avoid a lookup within the complete hash table. In fact, the modelization approach we are proposing aims at evaluating the operation of an optimized NF operating in average conditions.

MPLS Switch

We evaluate the MPLS Switch model presented in Section 2.3.1 using MPLS over Ethernet frames. As in the previous case, we perform 2 different tests, one where each source sends traffic with only one MPLS label, and a second test where each source sends packets marked with 10 different MPLS labels. A rule matching each label present in the source traffic is added in the switch before the test begins.

The results of the tests and the estimate plotted in Figure 2.12 show that in both scenarios the measured throughput is below the estimated value. Since MPLS packet processing is computationally very similar to basic forwarding, the model estimates a *16.76 Mpps* throughput that is close to the *17.47 Mpps* estimate for basic forwarding. On the contrary, the measured throughput for the MPLS switch when operating with multiple labels (in a scenario comparable to the basic forwarding tests) is well below (about one third) the one obtained with basic forwarding, which hints to a poor optimization of MPLS processing in the software switch implementation. This

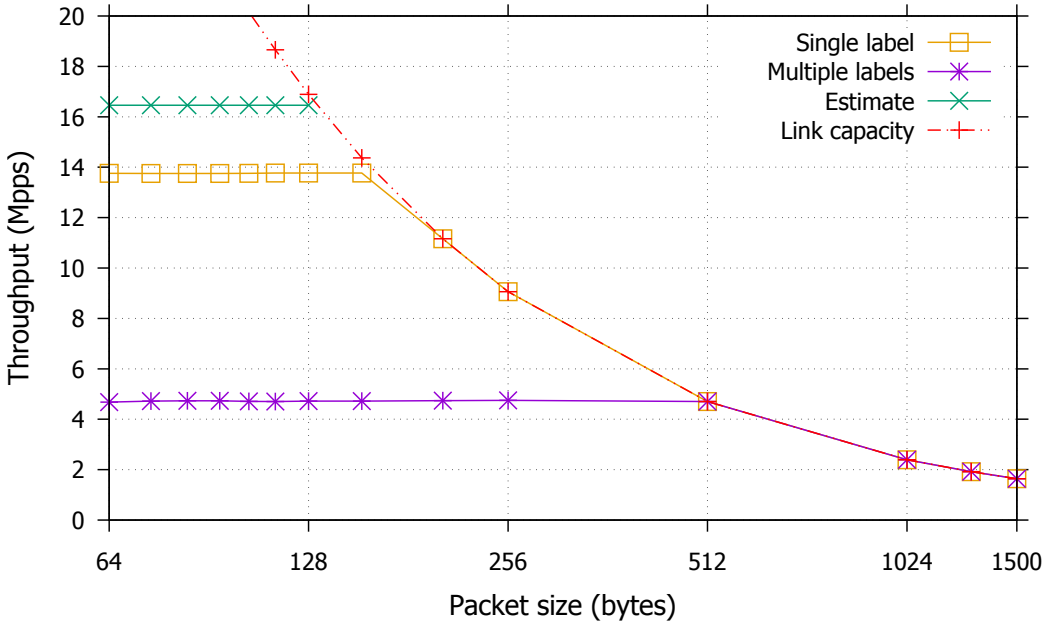


Fig. 2.12 MPLS switch performance.

might be related to the fact that forwarding based on MPLS labels was added to Open vSwitch relatively recently, hence the code is not as mature and optimized as the Ethernet address-based forwarding one. The large difference between the single and multiple labels tests shows that caching is playing an important role and in a real scenario, with traffic with multiple different labels, the software switch performance takes a significant hit (being almost one third of the case that takes advantage of caching).

2.4.2 Broadband Network Gateway

We run our tests on the BNG platform provided by Intel DPPD [18] on the hardware platform presented in Figure 2.2. This platform has been upgraded with one additional Intel 82599ES network card with 2x10Gbps Ethernet ports, given that the software requires 2 ports connected to the access network and 2 ports connected to the core network. The Intel Packet pROcessing eXecution Engine (PROX) is run on a second machine with the same hardware characteristics as a traffic generator. The test is run using the Intel Dataplane Automated Testing System (DATS), which controls one instance of PROX running on the tester machine to generate and to analyze the traffic and one instance of the BNG on the other host. DATS generates a realistic

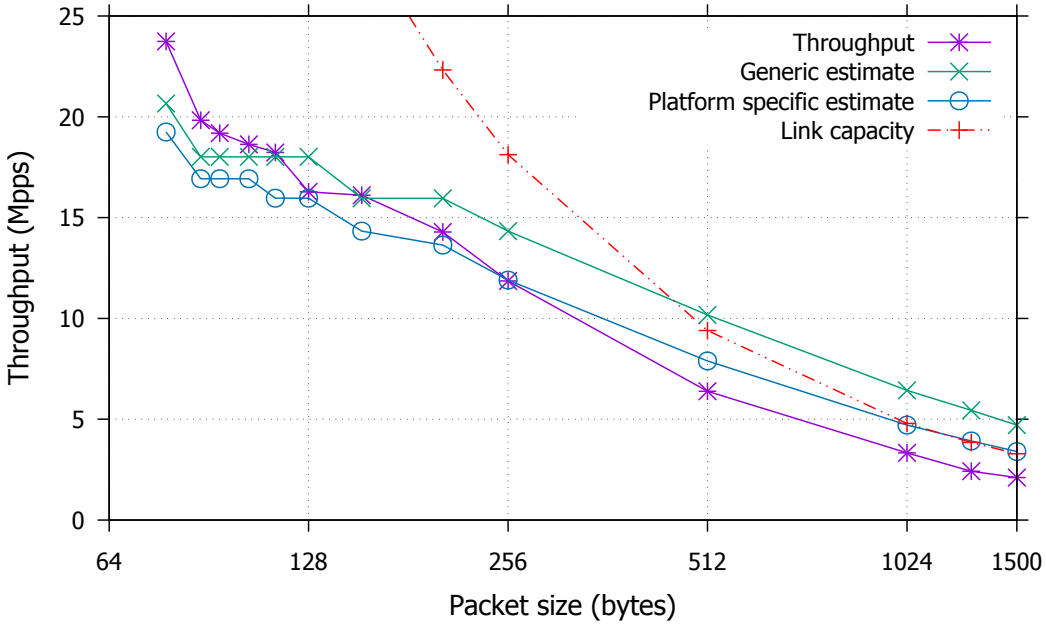


Fig. 2.13 Broadband Network Gateway performance.

workload simulating traffic from 32K users per port and with 8K possible routes. The test is executed 10 times and the averaged results are presented in Figure 2.13, where they are compared with the estimate devised in Section 2.3.2. Since DATS reports the aggregated throughput, corresponding to the total number of packets per second processed by the BNG, the plotted estimate is the average of the throughput estimates in the 2 directions.

The BNG software spawns 4 load balancer threads (one per interface) distributing the traffic among 6 packet processing threads. Therefore in the estimate we consider that 6 cores are dedicated to packet processing. Thanks to the parallel execution the CPU could theoretically process up to *42.5 Mpps* with 78 byte packets (the smallest packet generated by PROX). Hovewer, given the considerable number of main memory accesses required by the BNG to process a single packet, our model concludes that the overall throughput (for small packets) is limited by the memory latency and only *23.74 Mpps* can be processed when packets are 78 byte long. As shown by the `generic estimate` line plotted in Figure 2.13, the model is quite accurate in estimating the performance for small packets, with a 7% average error for packets up to 204 bytes. However, the model is less accurate for larger packet sizes. This is due to a side effect of the distributed execution of the NF.

The hardware platform used in the experiments has 2 processors and 2 NICs, each NIC connected to the socket of one of the processors. The DDIO mechanism considered in Section 2.2.2 when mapping the `I/O_mem(hdr, data)` EO onto the hardware platform, moves each packet received through a NIC to the L3 cache of the processor it is connected to. When a packet processing thread running on the other processor executes the `I/O_mem(hdr, data)` EO on a packet stored in the L3 cache of the other processor, its cost is different than the one presented in Section 2.2.2 because the processor must read the packet from the main memory and load it into its own L1/L2 cache before starting processing it. As a result, in this case the execution of the EO requires:

$$30 + 5 * \lceil \frac{hdr + data}{64B} \rceil \text{ clock cycles} + \lceil \frac{hdr + data}{64B} \rceil \text{ DRAM accesses}$$

Note that the BNG needs to process the whole packet (i.e., `hdr+data` bytes), not just the header, in order to compute the GRE checksum.

Considering that that the 4 load balancer threads are uniformly distributing packets on the 6 packet processing threads and that the traffic load on the two interfaces is the same, there is a 50% chance for a packet not to be in the L3 cache of the processor running the corresponding processing thread, which is taken into account in plotting the `platform specific estimate` line in Figure 2.13. When compared to the `generic estimate`, it provides a more accurate throughput estimate for larger packets that cause a non-negligible wait time for the processor retrieving them from main memory.

2.4.3 Concluding Remarks

The comparison of experimental results with the estimates produced by our model presented in this section shows that software NF performance, and consequently the modeling accuracy, are heavily affected by multiple quite specific factors, such as:

- The effectiveness of caching mechanisms realized in both the execution platform (e.g., processor cache) and the software implementation of algorithms and data structures (e.g., the microflow cache). Cache deployment largely

increases the performance variability, which cannot be captured by a model since per-packet cost strongly depends on the traffic runtime characteristics.

- The implementation of the NF. Our performance estimation approach is well suited to NFs designed to perform a specific, well-defined, packet processing operation at high speed. In this context, the Intel BNG proved to be a valid use case for which the model provides a good estimation, being able to identify the per-packet processing cost and the aspects limiting the maximum throughput. On the contrary, general purpose implementations based on a generic, configurable pipeline to process packets in multiple ways are not well modeled by our approach, as shown by the tests on Open vSwitch. To provide programmability and flexibility, general purpose implementations might perform for each packet a number of operations not specifically needed by the required function. Our experiments showed that only when Open vSwitch is configured to perform the simplest supported operation (i.e., basic forwarding of Ethernet frames), the performance is more predictable and correctly modeled with our approach.

- Parallel execution of operations. Our approach is not meant to model the interactions and dependencies among components running in parallel on different processors. As shown by the case of the BNG running 6 parallel threads on 2 processors, the model had to be specialized to take into account the specific scenario.

In summary, our experimental evaluation demonstrates that a generic model cannot fully capture all the aspects that can affect the performance of an NF, which could be achieved only by delving into a level of detail that would make the model extremely detailed and specific of a given implementation and instantiation for execution on a specific hardware platform. Hence, the model cannot be expected to estimate the performance of an NF with high accuracy. Such relatively loose estimate is anyway not worthless and has at least two very valuable applications: (*i*) in support of VM scheduling and VNF orchestration in cloud environments and (*ii*) as a reference performance upper bound in both the design and improvement of a NF software implementation.

Moreover, the proposed NF model can also be mapped on hardware implementations, in which case we expect the performance to have less variability and consequently the model to provide a more accurate estimate.

2.5 Related work

The work described in this chapter was initially inspired by [13] that aims to demonstrate that the Software Defined Networks approach does not necessarily imply lower performance compared to purpose-built ASICs. In order to prove it, the performance of a software implementation of an Ethernet Provider Backbone Edge Bridge is evaluated. The execution platform considered in [13] is a hypothetical network processor, for which a high-level model is provided. Unlike our work, the authors do not aim at providing a universal modelization approach for generic network functions. Rather, their purpose is to leverage the usecase of a specific sample network function to demonstrate that, even for very specific tasks, the NPU-based software implementation offers performance only slightly lower than purpose designed chips.

[22] presents a modeling approach for describing packet processing in middleboxes and the ways they can be deployed. The approach is applied to a NAT, an L4 load balancer, and an L7 load balancer. The proposed model is inherently different from ours in that it is not aimed at estimating performance and resource requirements, but it rather focuses on accurately describing functionalities to support decisions in the middlebox deployment.

Cloud platform management solutions that take into account the performance of the network infrastructure when placing VMs [23–25] could greatly benefit from a VNF performance estimate. For example, [25] describes the changes needed in the OpenStack software platform, the open-source reference cloud management system, to enable the Nova scheduler to plan VM allocation based on network properties and a set of constraints provided by the orchestrator. We argue that in order to enforce such constraints, the orchestrator needs a VNF model like the ones generated by the approach presented in this chapter. However, the presented methodology cannot be applied as such to VNFs because the additional overhead introduced by virtualization must be considered. A few works addressed this specific aspect. [26] presents a generic model to predict performance overheads on various virtualization platforms, based on the evaluation of the most influencing factors, such as CPU

scheduling and resource overcommitment, while in [27] the virtualization overhead is estimated with focus on the impact of sole resource contention. Resource usage of virtualized applications is addressed in [28] by means of regression models, starting from benchmark results. While these studies offer ways of estimating virtualized application performance, when considering an NFV environment it is essential to take into account the overhead related to the virtual switch in the hypervisor, which uses a relevant share of processor time to forward traffic to and from VNFs. Our modelization approach can be applied to devise an estimate of the resources required by the virtualized network function and inter-VMs traffic steering, thus enabling a more accurate VNF performance estimate.

2.6 Conclusions and future work

In this chapter we presented a unified modeling approach aimed at performance estimation of Network Functions when executed on different platforms. Starting from the identification of the most relevant operations performed by the NF on the majority of packets, the presented methodology allows to define a platform independent model of such a NF. The model can then be automatically mapped to the target execution platform, leveraging the characterization of hardware performance. This methodology is especially helpful in planning VNFs placement and resources allocation, and is valuable for integration of middleboxes in an NFV infrastructure.

The presented experiment results show that the proposed modeling approach provides a way to obtain a usable, even though loose, estimate of NF performance, especially for single-purpose, highly optimized, software implementations. The results show also that a very accurate estimation cannot be obtained without taking in consideration characteristics of the traffic. We claim that the proposed modelization approach can be valuable for those application where the traffic profile is not known a priori, such as VNF scheduling and orchestration. Moreover, the model can be fine-tuned at runtime with the support of traffic and performance monitoring to adapt to the traffic profile. We plan, as future work, to integrate the modelization methodology with online refinement, leveraging live performance monitoring. We also plan to investigate the application of the modeling approach to estimate the performance of hardware NFs and to evaluate the performance cost of traffic steering in a cloud computing enviroment.

Chapter 3

Enabling NFV Services on Resource-Constrained CPEs

3.1 Introduction

While telecom operators need to have a flexible infrastructure that can rapidly and efficiently provide dedicated, on-demand network services, so far this possibility was available only by deploying dedicated middleboxes in the network, which is known to be complex and costly. The ETSI NFV [10] architecture could be a possible answer to this problem, as it proposes to exploit virtualization techniques, typical of cloud computing, to instantiate Virtualized Network Functions (VNFs) in the operator data centers with unprecedented agility.

While cloud providers can count on centralized data centers encompassing mainly homogeneous servers, telecom operators feature an existing widely distributed infrastructure made of heterogeneous devices. In particular, although we can see clear benefits by integrating current Customer Premise Equipment (CPE) in the Network Functions Virtualization (NFV) infrastructure [30], those devices are usually based on low-cost hardware that cannot support VNFs under the form of virtual machines.

However, we can note that most CPEs are based on the Linux operating system, which includes (hence it can potentially execute) a broad set of existing software

The content of this chapter has been published in [29].

network functions (e.g., firewall, NAT, virtual switch, etc) running on the bare hardware.

To exploit this capability, we propose a software architecture that integrates existing CPEs in an NFV domain, leaving complex VNFs in the data center while simple *Native Network Functions (NNFs)* are executed in the CPE with low hardware resources, especially on the Home Gateway, hence combining the benefits of the cloud with the locality of the services running on local CPEs.

NNFs rely on *native capabilities*, i.e. software components that are already available on the CPE and that can be executed directly on the host operating system. In particular, the concept of "native" involves not only regular and built-in network functions (such as a virtual switching instance), but also elements (e.g., the Linux *iptables* module) that can be exploited to build network services (e.g., a firewall), as well as possible hardware accelerators that may be available on the node itself. As a result, native functions lead to significant improvements, in terms of memory consumption, storage requirements and start-up time, compared to existing technologies (LXC, Docker, VMs), enabling the execution of network functions even on resource-constrained devices. Moreover, NNFs can exploit hardware components (e.g. crypto hardware accelerator, integrated L2 switch) already available in the CPE, reducing power, space and required compute resources, as well as increasing security and performance.

Our solution enables an NFV orchestrator to optimize the scheduling of the NFs by starting the services that require network functions close to the end user (e.g., IPsec terminator, low-latency services) directly on the user CPE, while other components of the same service (e.g., the NAT module) are executed in a remote data center. This requires our architecture to define an abstraction that allows the orchestrator to understand the capabilities of the underlying infrastructure domain, and that can handle the lifecycle of each network function independently from its actual implementation. Furthermore, a reasonable security model has to be defined in order to support multi-tenancy for NNF as well, as the nice properties in terms of security and isolation guaranteed by traditional hypervisors are not available in our context.

The rest of this chapter is organized as follows. The next section examines related works, Section 3.3 describes the technologies this work builds upon. Section 3.4 presents and describes Native Network Functions with their abstraction. Experimen-

tal results that validate our proposal are shown in Section 3.5, followed by some final considerations and conclusions in Section 3.6.

3.2 Related Work

The necessity to introduce more flexibility in CPEs serving home/small office customers has increased over the years and has become evident with the emerging NFV paradigm. In fact, a recent trend consists in moving (part of) the CPE functions in the data center with the so called virtual CPE (vCPE) such as in [31, 32]; a minimal hardware appliance is left at the edge of the network, while (most of) the intelligence is moved to the cloud and implemented through virtual functions. An intermediate step toward a fully virtualized CPE is proposed in [33], which is based on the architecture defined by the Home Gateway Initiative industry alliance[1]. This architecture is highly modular and implements the different CPE functions as Java OSGi bundles, which can be dynamically loaded/discarded on demand. The *Surrogate vNF* proposed by the paper extends this paradigm by defining a set of "proxy" OSGi functions that keep compatibility with the existing architecture while delegate most of the processing to a companion VNF running in the cloud. However, the above solutions require excellent connectivity between the customer premises and the data center, and may introduce excessive delay for some latency-sensitive services. Furthermore, although in principle NFV enables a telecom operator to orchestrate its services by exploiting the resources offered across its entire network infrastructure, the vCPE approach cannot exploit resources that may be available on the CPE itself as VNFs are moved to the cloud.

Edge-based services are proposed in [34], which exploits eBPF programs to create a programmable data path in the CPE while the control plane is kept on the cloud. The CPE is able to handle locally the traffic, hence guaranteeing its operations also in case the connectivity toward the cloud is lost. Although this solution is very efficient, the eBPF virtual machine is not Turing-complete and cannot support even simple programs (e.g., string matching) that are rather common in network services. Considering that the CPE is usually resource-constrained, [35] proposes an optimization model that is able to select the best VNF among a set of possible choices, hence optimizing the cost of the VNFs deployed on CPE. However, they rely

[1] http://www.homegatewayinitiative.org/

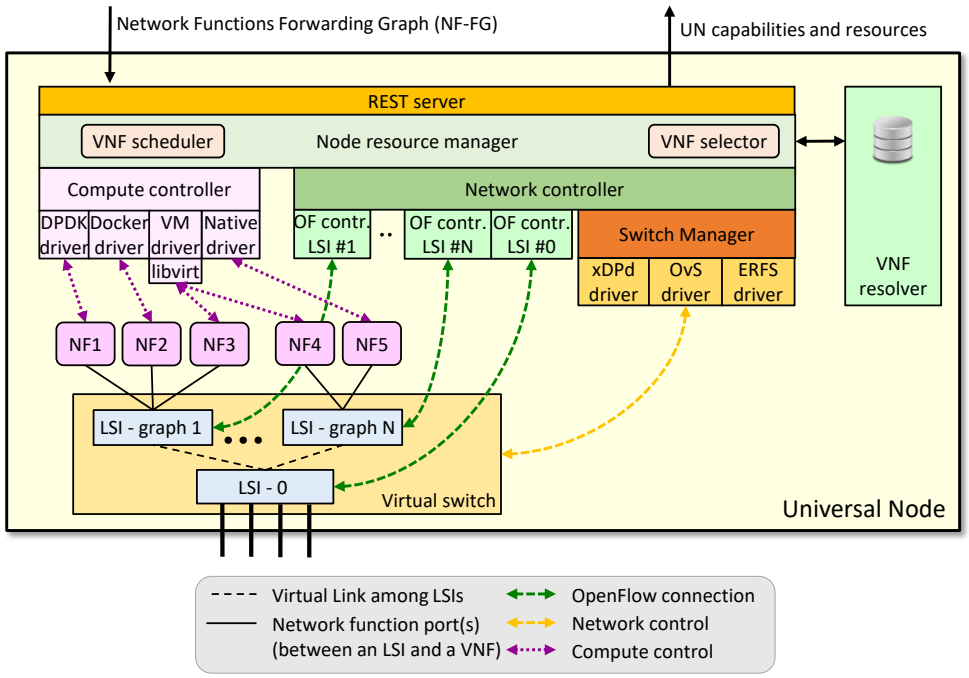


Fig. 3.1 Architecture of the Universal Node.

on the existing technologies for the VNF implementation such as Linux containers or virtual machines, thus being orthogonal with the idea proposed in this chapter.

To summarize, current NFV-compatible solutions do not support local processing in the CPEs, while more flexible CPE architectures are still limited in terms of supported features and are not compliant with the NFV world. Our proposal aims at achieving both objectives, namely NFV compatibility and arbitrary traffic processing in the CPE, while still supporting possible VNF running in the cloud, if needed.

3.3 Background

The architecture proposed in this chapter, depicted in Figure 3.1, is an extension of the *Universal Node* (UN) [36] developed in the EU UNIFY project [30].

According to the ETSI NFV terminology, [37] the UN includes different components of the reference architecture, namely an NFV orchestrator (NFVO), a Virtual Infrastructure Manager (VIM), and an NFV infrastructure instance (NFVI). The UN is a single box, e.g., a server, featuring a control plane able to jointly orchestrate net-

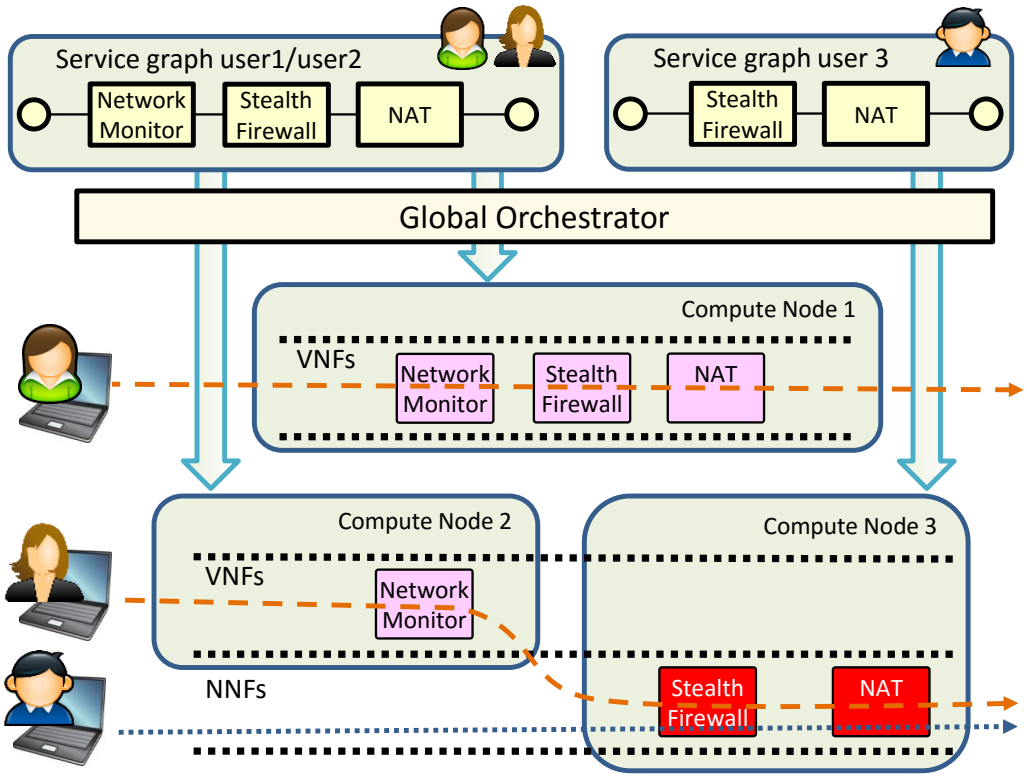


Fig. 3.2 Service instantiation of a graph.

work and compute resources. Moreover, it supports multiple execution environments and different virtual switches, and advertises functional capabilities (e.g., capability to execute a NAT service) instead of infrastructure-like information (e.g., KVM execution environment, available memory, etc.). The UN is a tiny infrastructure domain that exploits locally available information to optimize the service evaluating local resources/constraints, such as assigning VNFs to the best CPU cores.

Although service-layer requests (e.g., a new service graph that has to handle the traffic of a specific user) can be accepted by the UN orchestrator, they are usually issued to the global orchestrator because of its capability to leverage the resources available on different domains and to coordinate the deployment of the composing elements (e.g., VNFs) across the whole infrastructure. The global orchestrator, in turn, is able to split a service graph into a set of coordinated sub-graphs, which are passed down to the selected infrastructure orchestrators, such as in Figure 3.2.

Upon accepting a new service request from an overarching orchestrator that is in charge of the global deployment of the service across multiple infrastructure domains,

Table 3.1 Network abstraction in the UN

Function	Description
createLSI()	Create an LSI
deleteLSI()	Remove an LSI
createPort()	Create a port connected to a NF on an LSI
deletePort()	Remove a port connected to a NF from an LSI
createTSRule()	Generate a new traffic steering rule in an LSI
deleteTSRule()	Remove an existing traffic steering rule from an LSI

the UN can either deploy exactly the VNFs requested by the global orchestrator or, in case "generic" VNFs are chosen (e.g., a generic firewall instead of the one of a specific manufacturer) it relies on an additional component, the VNF resolver, to select the best implementation available matching the service request. Furthermore, it creates a new Logical Switching Instance (LSI) to properly steer the traffic among the selected VNFs.

3.3.1 Network abstraction

The network controller of the UN manages the networking paths among the deployed VNFs through multiple levels of LSIs: a base LSI-0 and a set of LSI-N (where $N \geq 1$), each one in charge of a different deployed graph (Figure 3.2). The first (LSI-0) dispatches the traffic from the physical interfaces of the machine to the LSIs of the other graphs. The additional LSIs create the traffic steering paths between the VNFs that belong to that graph. Each LSI is managed by a separate embedded OpenFlow controller, provided by the UN, that dynamically inserts the proper rules in its flow table(s).

A switch manager module can control different types of virtual switches by means of the set of primitives listed in Table 3.1 and implemented by each technology-specific driver. Basically, the abstraction allows to (*i*) create/delete an LSI, (*ii*) create/remove a port on the LSI that is connected to a NF, and (*iii*) create/remove a traffic steering rule between VNFs or ports. This allows to replace a generic virtual switch implementation with an hardware-accelerated one, without any impact on the rest of the software.

Table 3.2 Compute abstraction in the UN

Function	Description
createNF()	Allocate the resources to start the NF and create a shadow (local) copy of the NF image (if needed)
startNF()	Attach ports to the NF, and starts the NF
stopNF()	Stop the NF, without deallocating resources
updateNF()	Update the NF while running, e.g., by removing or hotplugging new network interfaces
deleteNF()	Release the resources (memory, shadow disk image) allocated to the NF
pauseNF()	Suspend the NF execution (for possible migration)

3.3.2 Compute abstraction

The compute controller is responsible for the VNF lifecycle management, such as instantiate, terminate and update a VNF. This is achieved by defining a common compute abstraction (Table 3.2) that is generic enough to be applicable to any type of execution environment. This abstraction is implemented by a set of drivers, each one in charge of a specific execution environment technology (e.g., VM, Docker, DPDK process) with the associated required parameters. For instance, the plugin that manages the KVM hypervisor creates on the fly the proper XML file required by the `libvirt` library for the VM instantiation when the `createNF()` call is invoked.

Each compute driver needs also to support different types of interfaces (e.g., `dpdkr`, `virtio`, etc.), according to the specific execution environment, as each execution environment supports only a subset of the available port types. In this respect, the compute controller needs to be coordinated with the network controller in order to attach the VNF ports, according to the required technology, to the existing software switch.

3.3.3 Northbound interface

The northbound interface of the UN is bidirectional: the downstream direction is based on generic service graphs that have to be instantiated on the node, while the upstream direction is used to export the information needed by an overarching

orchestrator and that is used to properly instantiate the requested service across different infrastructure domains.

The UN exports three types of information to such an upper layer orchestrator.

- **Functional capabilities** represent the ability to execute a specific network function, such as a NAT or firewall service, optionally with some specific characteristics, such as the capability to handle high amount of traffic (e.g., because it can exploit an hardware accelerator available in the node).
- **Infrastructure-level capabilities** refers lower-level characteristics, such as the CPU architecture, the ability to execute generic VMs or Docker containers, etc.
- **Available resources** refer to the about of unused hardware resources, such as the amount of free memory or the presence of a hardware accelerator.

The capability to advertise functional capabilities is a unique feature of the UN and it represents also the main reason we can bring the concept of Native Network Functions in this environment. In fact, an overarching orchestrator that operates based on functional capabilities will not decide the actual VNF implementation to be used, but it will only tell the underlying domain (e.g., the UN) to start a specific network function, leaving to that domain the decision about the specific VNF flavor (e.g., VM, Docker, etc) to be used. In turn, the UN will delegate this decision to the the VNF resolver.

3.4 Native Network Functions

This section introduces the concept of Native Network Function, i.e. a data-plane processing component that exploits capabilities natively present on the compute node and cannot be exploited by current NFV solutions. Our architecture allows NNFs to work alongside traditional VNFs, giving the possibility to improve overall network performance without losing the flexibility guaranteed by the NFV approach.

3.4.1 NNF model and VNF template

The first step towards the integration of NNFs in an NFV architecture is to define a model carrying all the information needed to properly execute them on a compute node. In fact, besides all the information required for the execution of a generic VNF (e.g., number of ports, port types), each NNF requires some additional properties to be satisfied in order to run on a compute node, namely *dependencies* or *requirements*. Those might refer to software packages (e.g., executables, libraries) available on the compute node that are already installed and that are required for the NNF to operate.

In addition, our model considers also information regarding the *status* of the allocated function, telling about current configuration and resource used by the function. This data is needed in order to be able to release the resources used by the NNF when the function stops, while in traditional VMs resources are freed along with the deletion of the VM.

In order to cope with this data, we extended the *network function template* to keep both VNF general attributes, common for all types of network function, and NNF-specific information. Figure 3.3 contains an excerpt of an NNF template representing a native firewall, which shows the properties of the function. In particular, it exploits *iptables* as a native capability and also supports multi-tenancy. The function handlers that will be used by the compute controller to drive its life-cycle (e.g., start, modify and stop) are available at the given URI with a specific format. The template also specifies basic I/O and network configuration of the function, information needed for driving the other NF types as well, not shown in the example.

3.4.2 The native compute driver

After receiving the VNF template, the compute controller has to control the native function by using the abstraction described in Section 3.3.2. The native driver will download the function using the URI specified in the VNF template, which points to a `.tgz` file. The above archive is a very compact file that includes a set of bash scripts that are called to perform the actions listed in Table 3.2, such as starting a new instance of the NNF, updating, stopping and all the other actions that are required in the VNF lifecycle management. As evident, the support for bash is the only requirement for running a NNF, which, in turn, enables native functions to

```
{
    "name" : "firewall" ,
    "uri" : "http://repo/native/firewall.tgz",
    "vnf-type" : "native",
    "multitenancy" : true,
    "dependencies" : {
        "capability": [ {
            "name" : "iptables",
            "type" : "package" ,
        } ],
    },
}
```

Fig. 3.3 Excerpt of the template of a firewall NNF.

be seamlessly deployed on machines with different CPU architectures. The files required for the execution of the NNF are:

1. a start script for the NNF instantiation;
2. a stop script to stop the NNF and to free the related resources;
3. the update script to update the NNF at runtime;
4. any other file that will be used for the configuration of the network function.

All the scripts that define the NNF are called by the plugin of the native function, which manages its life-cycle.

3.4.3 I/O model

In the traditional NFV framework, the traffic steering among the VNFs is carried out by a virtual switch that forwards packets according to the rules given by a network controller. Each VNF is provided with a certain number of virtual network interfaces that correspond to its ports, connected to the virtual switch.

In order to seamlessly support the execution of NNFs, the same I/O model must be repeated and therefore each NNF should be connected to the virtual switch with the appropriate number of ports. In this way, the network controller can create virtual ports for the NNF as well as for the VNF, without any modification to its logic.

In the NNF case, these ports are implemented as virtual Ethernet (veth) interfaces assigned to a network namespace on which the NNF is executed. As such, each NNF sees its own network interfaces that can use to retrieve/send its own specific network traffic.

3.4.4 Isolation model

Differently from current virtualization technologies that natively support an isolation model for the instantiated VNFs, the NNF driver needs to explicitly implement a layer that provides at least some form of isolation of the NNF against the rest of the system.

The NNF driver leverages the Linux namespaces by creating a new *network* namespace before running an NNF, adds to it the virtual network ports required by the function, and then launches the NNF inside the namespace. As a result, the NNF sees only the incoming traffic sent by the virtual switch to its veth. The name of the namespace is unequivocally related with the graph and the function name, thus avoiding possible collisions. At the end of the execution of the NNF, the namespace is deleted by the native driver and all the other related resources are freed.

Differently from Linux containers that exploit *all* the different types of namespaces available in the Linux OS, NNFs use by default only the network one in order to guarantee network isolation between different NNFs. A more sophisticated isolation model, leveraging multiple namespaces that can be activated on demand (based on the requirements of the tenant, the infrastructure owner, and NF), is currently in progress.

3.4.5 Multitenancy

In a traditional NFV architecture in which each VNF runs on a distinct VM, multitenancy is an intrinsic property of the execution model. In fact, multiple instances of the same VNF can always be launched while traffic steering primitives can set the proper flow rules in the software switches to create the correct traffic steering paths among VNFs.

Supposing that a NNF can be instantiated multiple times, multitenancy is achieved by encapsulating multiple instances of the NNF in dedicated namespaces whose

virtual interfaces are connected to different ports of the software switches. On the contrary, if a NNF does not support multiple instances running at the same time (e.g., it relies on an hardware coprocessor that cannot be shared among functions), multitenancy should be managed by means of an ad-hoc marking mechanism that allows the NNF to distinguish between traffic belonging to different service graphs. Moreover, the NNF should create multiple internal paths to process the above multiple traffic streams disjointly.

3.4.6 Security considerations

Launching a native function, hence a script running on the bare hardware, offers less protection than starting a software in a VM or in a container, which can leverage the additional protection shield provided by the hypervisor or the container execution engine. For instance, little protection exists to limit the resources used by native functions, e.g., in terms of CPU/memory consumption or the number of occupied CPU cores. Although the impact of the above problems could be limited by turning on some addition Linux mechanism such as `cgroup`, this complicates the solution to the extent to which other alternatives may be more appealing, such as replacing the NNF with a container-based implementation.

In any case no protection exists that prevents a VNF, which is expected to provide a given service (e.g., firewall), to behave differently (e.g., to launch an attack toward a remote host) and the current solution is simply to trust the creator of the application or the entity (e.g., app marketplace owner) that sells it. Therefore, although we acknowledge that the problem of determining whether a NF is malicious is emphasized in case of NNF because of their inferior degree of isolation, we feel that the problem is rather general and should require a more generic solution that guarantees, a priori, the goodness of the VNF, e.g., by means of novel software verification techniques.

In this respect, a possible direction for future investigation could consist in integrating remote attestation techniques [38] in our execution environment, exploiting an external machine to verify the correctness of the running software.

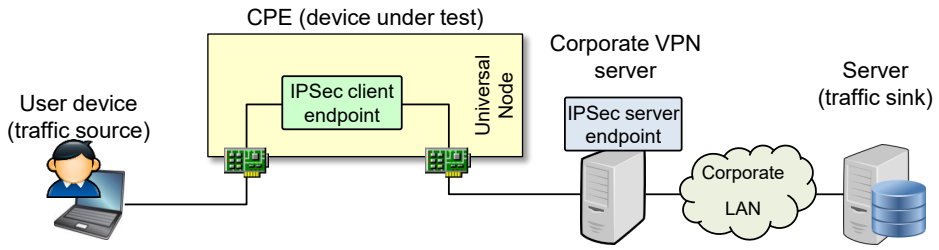


Fig. 3.4 Testbed used in the validation.

3.5 Validation

This section presents the results of a preliminary validation campaign with a transparent VPN access use case, i.e., when a user client located on a trusted local network (e.g., home) needs to connect to its corporate VPN server. In order to avoid the necessity to install the VPN client software on all user's devices (e.g., laptop, smartphone, etc.), the VPN client is instantiated on the user's CPE, hence providing secure access to the corporate network independently of the user device.

The testbed, shown in Figure 3.4, encompasses a client that generates the traffic, a CPE executing the IPSec client NF in charge of encrypting/decrypting the traffic, a VPN server with the opposite duty, and finally a traffic sink. All the four boxes are connected with point-to-point 1 Gbps Ethernet links; faster speeds are usually not available in low-end CPEs. Three powerful workstations were used respectively as traffic source, VPN server and traffic sink in order to avoid those machines to become the bottleneck, while different flavors of CPEs are used, namely a mid-range server, a business CPE based on the Freescale T1040 and a domestic CPE, all with the same version of the UN software, although compiled for the specific platform. The specific hardware and software details are listed in Table 3.3.

The UN was configured through its northbound interface with a very simple service graph, featuring an IPsec client NF connected to the LAN and WAN ports; the NF was based on the well-known Strongswan [39] software, configured to operate in IPsec tunnel mode (using IKEv2 to establish the security associations, AES-CBC-128 for the encryption and SHA1-HMAC for verifying the data integrity).

The use of different hardware platforms was coupled with different implementations of the same NF, whenever possible. The server-based CPE was tested with three

Table 3.3 Characteristics of the devices used in the validation

Machine(s)	Hardware and software characteristics
User device (source) Traffic server (sink) Corp. VPN server	Intel Core i7-4770, 32GB RAM, 500GB HD Linux Ubuntu 14.04, Kernel version: 3.16
Server CPE	Intel Core i5-3450S, 8GB RAM, 200GB SSD Linux Ubuntu 14.04, Kernel version: 3.19
Domestic CPE	Netgear R6300v2, CPU Broadcom BCM4708A0, 800MHz (2 cores), 128MB Flash, 256MB RAM OpenWrt 15.05, Kernel version: 3.18
Business CPE	Hawkeye HK-0910, Freescale QorIQ T1040, 1.2GHz (four e5500 cores), 64MB NOR Flash, 2GB RAM DDR3L-1600 Freescale QorIQ SDK V1.7, Kernel version: 3.12

equivalent network functions based on VM, Docker and NNF, while the business CPE and the domestic CPE supported the network function only as sofware-based NNF.

Our experiments took into consideration (*i*) the throughput between the two hosts and the associated CPU load during the experiment, (*ii*) the amount of RAM consumed, (*iii*) the NF image size, (*iv*) the amount of additional libraries required to start the requested execution environment in addition to the base Linux system (e.g., KVM/QEM for VMs) and (*v*) the time required to start the NF. The first two experiments leveraged the `iperf` tool installed on the source and sink machines, configured to generate two unidirectional TCP streams at the maximum speed. We set the packet size such that the MTU is not exceeded after the addition of the IPSec header, in order to avoid fragmentation. All experiments were repeated 10 times and averaged.

The throughput, in the second column of Table 3.4, shows that NNFs and Docker bring significant performance improvements compared to VMs because of the simplified architecture that does require neither the hypervisor nor the guest OS, where the NF is running. Their throughput is higher with a reduced CPU consumption as well. In this respect, NNFs and Docker show the same level of performance, as expected, given that they are based on the same technology (i.e., kernel-based processing in the host plus namespaces).

Table 3.4 Comparing different implementations of the IPSec client, on different machines

IPsec client implementation	Thr./CPU (Mbps/load)	RAM (MB)	NF image (MB)
1) Server CPE - KVM/QEMU	796 / 100%	390.6	522
2) Server CPE - Docker	1095 / 80%	24.2	240
3) Server CPE - NNF	1094 / 80%	19.4	5
4) Domestic CPE - NNF	57.2 / 100%	5	2
5) Business CPE - NNF	617 / 90%	1.9	3.7

The *memory occupation*, i.e., the amount of RAM required to execute the given NF and the execution platform, showed in the third column of Table 3.4, exhibits the same trend. In this case numbers can only be considered as qualitative measurements, as they may change considerably by tuning the NF in a different way, particularly for the VM case. In our test we created a guest OS with the default installation of a Ubuntu server 14.04, installing only the packages required for our VNF to work. As evident, the memory occupation is definitely higher in the case of VMs, while Docker and NNF are very similar, although they slightly vary according to the hardware platform under consideration. Note that Table 3.4 reports the application-level throughput, i.e., measured on the source/sink machines. Packets are extended with the additional IPsec headers required to create the tunnel, hence reaching, between the CPE and the IPSec server endpoint, an higher throughput.

The fourth column of Table 3.4 shows the *NF image size*, which confirms definitely the advantages of NNFs not only with respect to VMs, but also against Docker, as the image size is about two orders of magnitude less than its counterparts[2]. Moreover, this impacts also on the time required to download the NF image from a remote location, which is critical when the CPE is connected to the Internet through slow links (e.g., ADSL). An additional test was carried out in the host environment to measure the *additional disk size*, required in the host, to support the execution of the specific environment; due to the intrinsic limitations this was only possible on the server-based CPE. Starting from a clean installation of Ubuntu server 14.04 with default settings, we measured an additional 40 MB for the components required to execute VMs (i.e., KVM/QEMU) and 30 MB required to execute Docker containers.

[2]The image size of a NNF is merely the size of the NF software, compiled for the target platform

The above numbers confirm the advantages of the NNF with resource-constrained environments; in fact, the reason for not testing Docker on the home and business CPEs is the disk size limitation on those platforms.

Finally, we measured also the *time to start a NF* in the server-based CPE, being the only environment that can start all the NF types. The result showed about 3 seconds with VMs (which require starting the entire VM), 350 ms with Docker, and 727 ms with NNF; the baseline, i.e., the time required to launch the IPsec client on the base system without wrapping it in any NF, was 154 ms. This confirms, once more, the advantage of running applications in the host; the (relatively) high number with NNF is due to some implementation-dependent delay required to attach the network ports to the NNF and will be addressed in a future optimization.

In this preliminary evaluation we have considered a very simple service chain. We reasonably expect that more complex chains, composed of many NFs, would increase the advantages of NNFs over VNFs, given that resources usage, such as main memory, would become even more critical.

3.6 Conclusions

This chapter presents the idea of *Native Network Functions*, an NFV abstraction that allows to execute network functions even on resource-constrained devices by exploiting their native (both software and hardware) capabilities. Our preliminary validation campaign confirms that NNFs can be implemented over a reasonable variety of hardware, ranging from standard high-volume servers to business and domestic CPEs, with different hardware characteristics (CPU architecture and speed, memory size, etc.). Furthermore, NNFs can export existing hardware accelerators as network functions, hence enabling an NFV orchestrator to transparently take advantage from the superior efficiency of the hardware compared to pure software implementations.

Future work will aim at extending this approach to support traditional middleboxes as well (e.g., routers, switches, etc.), allowing their seamless integration in an existing NFV infrastructure.

Chapter 4

Enforcement of Dynamic HTTP Policies on Residential Gateways

4.1 Introduction

Modern residential gateways are widely deployed to provide broadband Internet access to families, small and medium-sized enterprises supporting a wide range of data rates, from a few Mbps up to 1 Gbps [41]. The architecture of residential gateways is characterized by special purpose hardware chips that forward packets at high speed at the data link layer, while general-purpose components, such as CPU and central memory, are usually employed for other operations that require more sophisticated processing. Since all the traffic directed to Internet hosts (i.e., outside the residential or corporate branch network) must pass through the residential gateway, it is the ideal appliance to apply traffic filtering. Hence, its processing capabilities, often underutilized, could be leveraged by Internet access service providers to offer such additional service to their customers. However, the limited computing and memory resources that residential gateways have by design make the implementation of new features working at wire-speed very challenging, particularly when complex operations such as parsing packets up to the application layer (a.k.a. Deep Packet Inspection or DPI) are involved. This is the case for many critical modern filtering applications, such as malware protection, corporate policy enforcement,

The content of this chapter has been published in [40].

parental control, advertisement block, that are based on inspection and filtering of Uniform Resource Locators (URLs). In fact, users access and exchange content mostly through mobile apps and web applications, both based on HTTP, which uses URLs to identify data objects to be transferred.

Currently, the above URL filtering-based services are most often operated in web proxies [42] or in end-user devices (e.g., laptop, tablet, smartphone), as a mobile app [43] or a browser plugin [44]. None of these solutions can guarantee that all the outgoing traffic is analyzed and filtered; in fact, a user can switch to a different device, disable the filtering software or change the client network settings in order to bypass a web proxy. The residential gateway is the perfect spot where to implement services that require all the web page requests to be analyzed. This would require matching URLs against large, dynamic blacklists, which far exceeds the limited hardware capabilities of this category of devices. For example, an effective parental control service, which is a valuable offer to residential customers, is based on a very large database of URLs that cannot be stored in the limited memory of common residential gateways (usually in the order of tens of MB). An additional challenge comes from the fact that the database must be frequently updated. Last but not least, URL matching cannot be limited to the hostname, but the entire URL should be considered because the same web server can host both appropriate and inappropriate or malicious pages. Hence, looking up a URL within a huge list of blocked resources exceeds the processing capabilities of a residential gateway, especially if it must be done for live traffic, which implies that the additional introduced delay must be limited.

This chapter presents *U-Filter*, an efficient solution to integrate a URL filtering service in a resource constrained device, such as a common residential gateway, leveraging a distributed architecture. A remote *policy server* in charge of keeping the URL database up-to-date provides a fast API that can be accessed through the network in order to establish if a request for a specific URL is allowed. It is reasonable that the above mentioned server is operated by a service provider (or the network service provider) and can rely on powerful hardware resources to serve multiple residential gateways with minimal response time. However, this architecture does not necessarily require the network service provider awareness and collaboration. The presented solution greatly alleviates the load on each residential gateway, even though it must still perform a limited form of DPI on outgoing packets to extract the URL from every HTTP request, and afterwards query the server in

order to determine the policy that must be applied. We adopt specific techniques to optimize this task and limit the latency introduced by the client-server interaction, striking a balance between the load they introduce and the limited resources available in residential gateways. Although the U-Filter design and the adopted optimizations are presented here in the context of policy enforcement on HTTP traffic, they offer a general solution for in-network policy enforcement suitable for a wide range of network protocols, thanks in particular to the decoupling of policy checking and enforcement phases, as detailed in Section 4.2.2.

This chapter is organized as follows. Section 4.2 presents the architecture of U-Filter, describing the design principles that led to our solution and the optimizations used to provide real-time policy enforcement on resource-constrained devices. In section 4.3 we evaluate the proposed solution by discussing its limitations and analyzing the additional delay introduced by U-Filter. We validate U-Filter in Section 4.4 through various experiments showing the impact on the user experience. Section 4.5 presents the state of the art of HTTP-level policy enforcement and Section 4.6 concludes the chapter with a discussion of future research directions.

4.2 Architecture and implementation

4.2.1 Operating principles

A typical deployment scenario of U-Filter is presented in Figure 4.1. A user surfing the web generates many HTTP requests that transit through her/his residential gateway. These requests are analyzed by U-Filter, which extracts the requested URL through a lightweight DPI algorithm. This allows to process line rate traffic with a small overhead for the residential gateway. Afterwards the HTTP request is released and can continue its journey towards the web server, while the URL is simultaneously sent to the policy server that provides the policy to enforce. This policy is enforced by U-Filter on the packet carrying the HTTP response by either blocking or allowing it. Thanks to the parallelization of the policy server and web server processing, this workflow greatly reduces the latency experienced by the user, making it comparable with the one that can be obtained with the same hardware without the service in place.

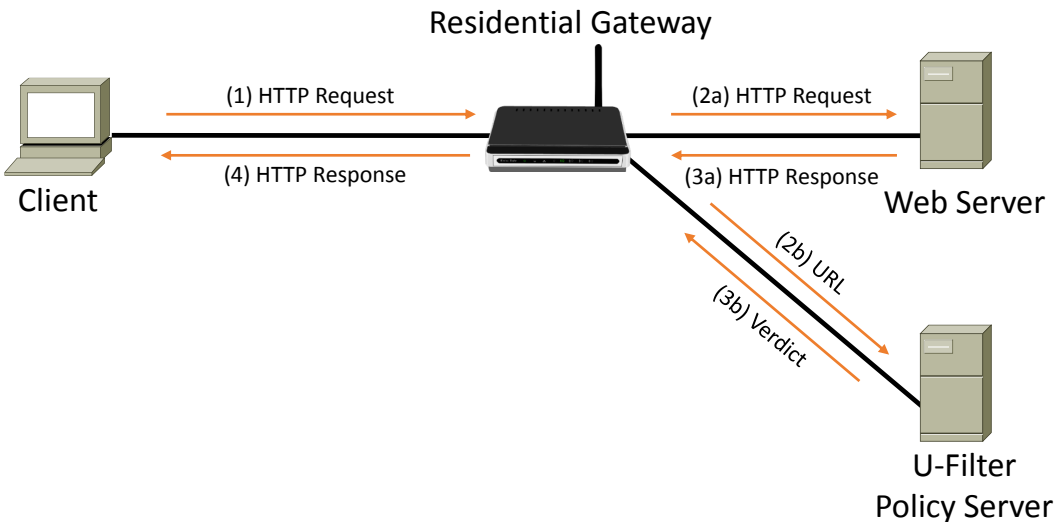


Fig. 4.1 U-Filter workflow.

4.2.2 Architecture overview and design principles

Our prototype has been built around three objectives. First comes **flexibility**, as it is essential to be able to enforce effective protection to end users in a prompt response to newly discovered threats. Second is **efficiency** since the system is targeted to resource-constrained devices. Third, we took care of ensuring an excellent **user experience**, hence limiting the impact of the system in terms of possible additional latency when inspecting traffic to apply filtering policies. The above high-level objectives have translated in the following four design choices.

Three-tier processing architecture

As shown in Figure 4.2, U-Filter includes (*i*) an *online module*, which sits on the data plane of the router and is mainly in charge of identifying (and extracting) requested URLs from network traffic (more details in Section 4.2.5) and apply the policy decisions on the return traffic, (*ii*) an *offline module* that queries a remote policy server to know whether such URL should be allowed or not (described in Section 4.2.6), and (*iii*) a *remote server* that implements the complex protection logic and returns a boolean value with the result of the classification, i.e., if the corresponding HTTP session handled by the online module has to be allowed or the URL is malicious and the response has to be blocked. The first two modules are built with efficiency in mind, while the latter allows to achieve the required flexibility.

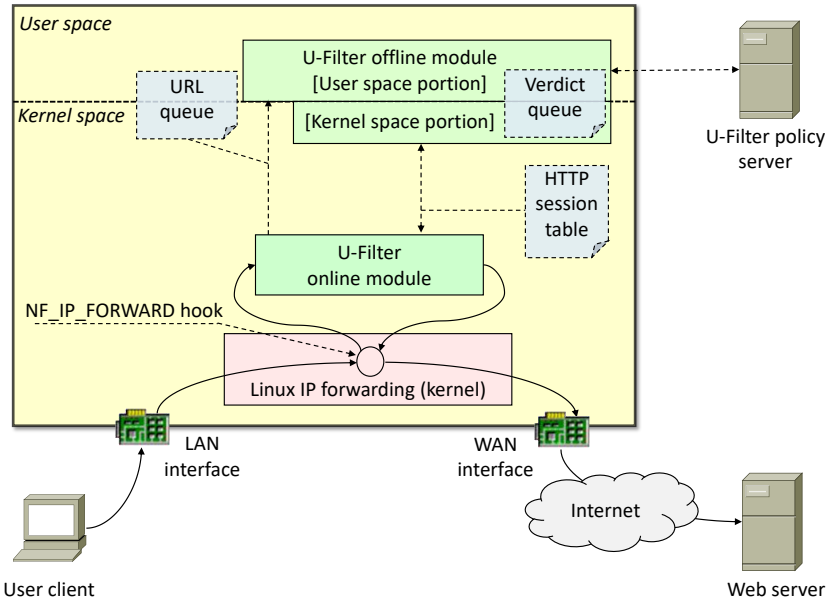


Fig. 4.2 U-Filter architecture.

The U-Filter online module is inserted on the path that packets being forwarded by the residential gateway take through the system. It leverages a hook provided by the `netfilter` [45] framework, as detailed in Section 4.2.3, available in the mainline Linux kernel, to enable interaction with the IP forwarding function. To achieve high performance, the online module is executed in the kernel space; this allows to avoid expensive kernel-to-user context switching and enables sharing the required data structures with the rest of the kernel (e.g., direct access to privileged memory areas), hence minimizing communication overheads. In fact, by working in kernel space, the online module can implement a *zero-copy* approach, since the data structure containing the packet data is not copied in the user space memory and is only referenced by the online module. On the other hand, the offline module is invoked a limited number of times compared to the online module because it operates only when a new URL is detected, but it requires more time to complete due to its interaction with the (remote) policy server. As a consequence, an asynchronous execution model is preferred for this module in order not to block the execution of the data path. This could be implemented as either a dedicated kernel thread or as a user-space process, which is the solution chosen in our implementation[1] because of the complexity of the tasks it executes and to avoid that any possible misbehavior

[1] In fact, a small portion of the offline module has to be implemented anyway in the kernel space, as shown in Section 4.2.6.

(or bug) can be propagated to the kernel, hence affecting the overall operation of the residential gateway.

The policy server can be executed on a remote host (or on a cluster of hosts for performance reasons), as its only interaction with the rest of the system is through a query/response protocol. A single policy server can be queried by offline modules running on multiple (remotely distributed) residential gateways. In our implementation, this interaction has been implemented with the ad-hoc dedicated protocol detailed in Section 4.2.7, but other choices (e.g., REST web service) are surely possible.

Decoupling policy verification from HTTP operation

As introduced in Section 4.2.1, policy compliance is verified without holding outgoing packets on their ride towards the final destination. This solution makes the system more complicated but much more efficient. In fact, keeping the HTTP request on hold until the arrival of the response from the policy server would add additional delay to the HTTP communication, increasing the Round Trip Time (RTT) of the HTTP connection and hence affecting the user experience. Vice versa, the U-Filter offline module checks the requested URL with the policy server during the normal HTTP RTT. A temporary entry in an HTTP session table is created by the online module in order to possibly hold a response from the web server received *before* the result of the compliance check arrives from policy server. While this allows packets to travel through the Internet also if they are part of a session that shall be stopped, the answer from the web server never reaches the user, effectively preventing possible unwanted data to reach the user's host.

Efficient memory usage

Efficient memory usage is a key problem because of (*i*) the limited amount of memory usually available in current residential gateways, and (*ii*) the bad effects in terms of CPU cache pollution when large memory structures (with sparse access patterns) are used. Several implementation choices have been adopted to ensure that memory is used efficiently. According to the best practice for kernel module development, all the memory used by the online U-Filter module is allocated at startup in order to avoid costly memory allocations at run-time, and the structures that are used

for the communication between online and offline modules are shared (using the proper primitives for mapping memory between kernel and user space) for better memory efficiency. Furthermore, all the helper structures (detailed in Section 4.2.4) make use of contiguous memory areas in order to improve data locality and, as a result, CPU cache efficiency, except for the packets that may need to be held temporarily by U-Filter (while waiting for an answer from the policy server), which have been allocated by other portions of the kernel and therefore are not under our control. Finally, the usage of additional memory is kept at minimum: (*a*) the data structure dedicated to the session table defines a "default" behavior that avoids storing accepted sessions, and (*b*) the number of packets held by the router while waiting for the answer from the policy server is limited to, at most, *one per session*, hence further reducing memory requirements.

Per-packet operation

This is known to be much more efficient than per-TCP session processing while, at the same time, reducing the latency required to extract application level information (namely URLs). In fact, the former can be based directly on the very efficient packet processing primitives available in the Linux kernel through the `netfilter` framework, instead of requiring a full-blown HTTP proxy, whose complexity is so high to make a kernel implementation problematic. Therefore, an additional overhead is added for moving all packets from kernel to user space, where a proxy is usually located, and then back to kernel for their transmission on the output interface.

As a downside, working on individual packets makes the system less robust against malicious attacks such as HTTP requests whose URLs are split across packets (possibly deliberately sent out of order). Such attacks could be spotted by adding lightweight, packet-based ad-hoc anomaly detection algorithms [46–48], which is outside of the scope of this work.

4.2.3 Netfilter

In order to gain access to live traffic, U-Filter leverages `netfilter` [45], a framework provided in the mainline Linux kernel that allows analyzing and modifying all the packets that are being received by the kernel. `netfilter` defines a set of hooks that correspond to different stages in the path packets take in the system.

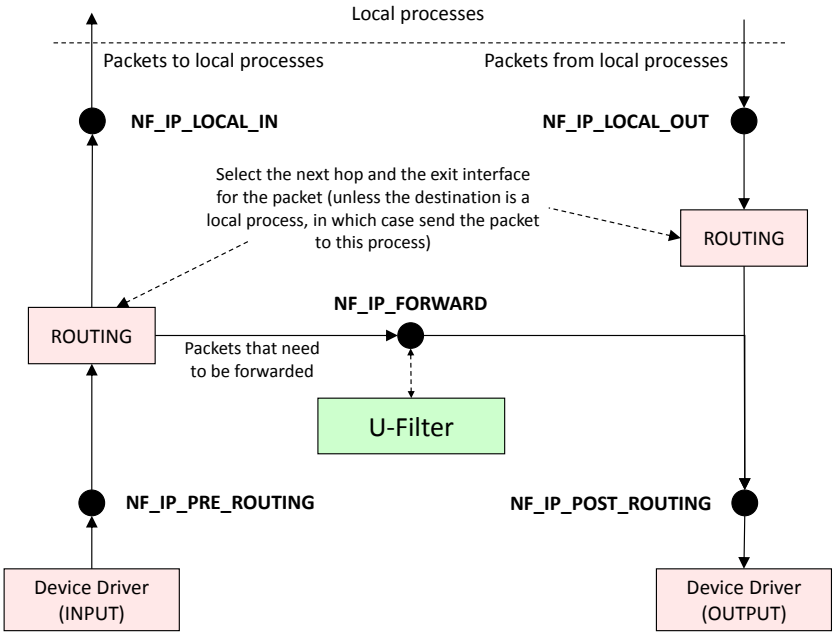


Fig. 4.3 `netfilter` hooks chain and U-Filter.

An application can register one or more callbacks linked to a specific hook; the corresponding callbacks are invoked whenever a packet passes through it. The callback receives a pointer to the system data structure containing the packet's data as a parameter, therefore it can read and modify the packet. Finally, the returned value instructs the system on whether the packet can continue its journey (`NF_ACCEPT`), or should be immediately dropped (`NF_DROP`), or should be diverted to a different (custom) processing pipeline (`NF_STOLEN`), which is useful if the decision about accepting/dropping the packet has to be postponed.

Figure 4.3 shows the possible paths taken by packets, together with the hooks that can be used to register callbacks. All the incoming packets are caught by the `NF_IP_PRE_ROUTING` hook, before being processed by the routing task; afterwards, packets addressed to the host itself are caught by the `NF_IP_LOCAL_IN` hook, while those traversing the host on their way toward the destination hit the `NF_IP_FORWARD` hook (where U-Filter is attached). The `NF_IP_LOCAL_OUT` hook catches packets sent by the host's local processes, while the `NF_IP_POST_ROUTING` hook catches all the outgoing packets, whether they are forwarded or locally generated.

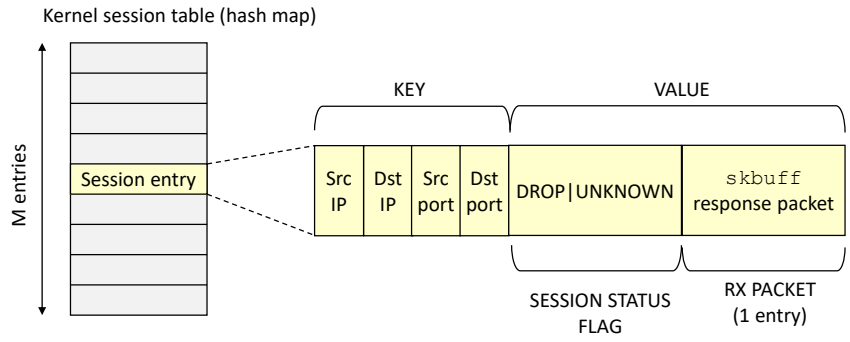


Fig. 4.4 HTTP session table, shared between online and offline modules.

4.2.4 Key data structures

The online and offline modules exchange data using three shared structures, as shown in Figure 4.2: (*i*) a hash map for the status of the policy for a given session, (*ii*) a queue for the URLs that have to be send to the policy server and (*iii*) a queue with the verdict received from the policy server. Each of the data structures is described in detail in the reminder of this section, while their usage will be discussed in the following sections.

The *HTTP session table* (shown in Figure 4.4) stores data regarding pending sessions. An HTTP session is considered pending when the HTTP request has been received, but either the HTTP response from the web server or the decision from the policy server are yet to be received. The hash map implementing the HTTP session table is allocated in kernel space and is shared between the online and offline module because the former needs to know (when an HTTP response arrives) whether a decision for an URL has been received, while the latter needs to know, when the verdict is available, whether an HTTP response is already waiting. An entry in the HTTP session table can be deleted as soon as both the HTTP response and the verdict from the policy server have been received.

The *URL queue* (shown in Figure 4.5) is shared between the online module and the offline module user space process, while the *verdict queue* (shown in Figure 4.6) is shared between the kernel thread and the user space process of the offline module. The two queues are managed according to a FIFO policy and the access to each queue is implemented with two pointers, pointing respectively at the first *free* and the first *full* slot.

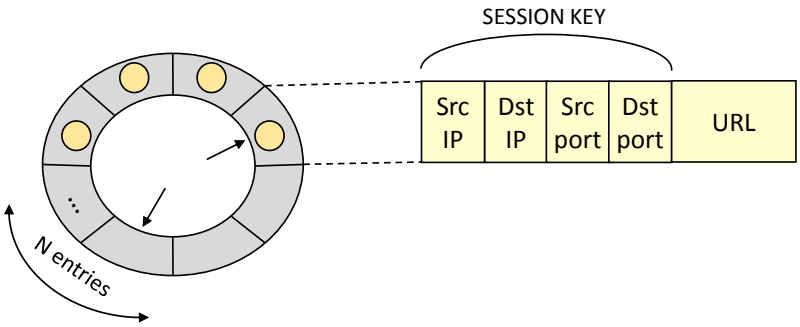


Fig. 4.5 URL queue, shared between the online module and the offline module user space process.

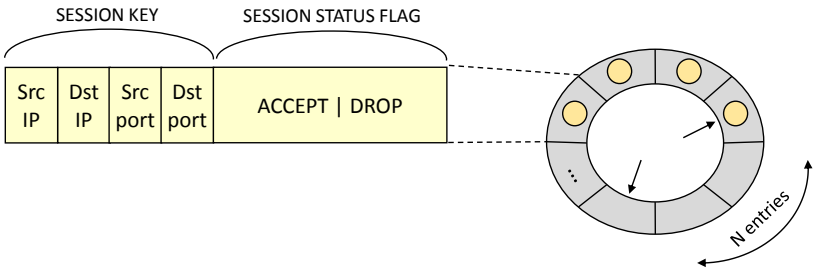


Fig. 4.6 Verdict queue, shared between the offline module kernel thread and user space process.

To correlate data in different data structures, an entry always contains a key made by the 4 tuple identifying the TCP session (later referred as session ID):

$$(Source\,IP, Destination\,IP, Source\,TCP\,port, Destination\,TCP\,port)$$

The addresses are the ones present in the HTTP request and are inverted in the corresponding HTTP response.

An entry in the URL queue contains also the URL that should be checked with the policy server, while an entry in the verdict queue contains a session status flag that assumes either `ACCEPT` or `DROP`, according to the policy to enforce. The URL is stored in some pre-allocated memory whose size allows containing a full-length HTTP payload (i.e., 1460 bytes), in order to avoid memory allocations at run-time. On the other hand, an entry in the HTTP session table stores as value a session status flag and a void pointer to a packet (`skbuff` structure, allocated by the operating system). The use of this pointer is detailed in Section 4.2.5. Differently from the verdict queue, the session status flag in the HTTP session table can assume either

UNKNOWN or DROP. In fact, entries corresponding to an ACCEPT policy are deleted as soon as the verdict is available in order to reduce the size of the hash table. Thus, in the HTTP session table the absence of an entry is considered as an ACCEPT policy.

As a further optimization to reduce the allocated memory, in our prototype the TCP session ID uses only the last byte of the source IP address, instead of the entire 4 bytes address, with no impact on the system proper execution. This optimization is correct in our environment, since domestics LANs usually adopt a 24 bits subnet, therefore all the clients have the same value for the first 3 bytes of the IP address. In general this is not valid for every deployment, hence the optimization should be adapted to the specific addressing plan in use.

4.2.5 Online module

The online module sits on the data path by intercepting all the traffic forwarded by the router through a callback registered on the NF_IP_FORWARD netfilter hook[2]. As shown by the workflow depicted in Figure 4.7, most of the processing occurs when an HTTP *request* or *response* is detected. For each packet, the module first locates the beginning of the TCP payload and then checks if that packet can be considered the *first segment* of an HTTP request or response by matching the beginning of the TCP payload against a few simple text strings, namely an HTTP method (i.e., GET, POST, PUT, etc.) in case of a request or a version string (i.e., HTTP/1.0 or HTTP/1.1) in case of a response. This classification method is far more reliable than checking the transport-layer port number, as investigated in [49]. All other packets, namely HTTP packets that are not the first of the request/response message (hence, do not match the signature), as well as non-HTTP traffic, are left to continue their way as the online module returns NF_ACCEPT to netfilter. Notably, since *all* TCP packets containing a valid payload are matched against the signature, this algorithm is able to intercept *all* the HTTP requests/responses that are issued within a connection in HTTP 1.1 persistent mode, not only the first one, as well as within HTTP connections terminated on a non-standard TCP port. This algorithm could raise concerns about the cost of inspecting all packets, as general DPI techniques are normally demanding in terms of computing resources. However, our algorithm does not perform a full-blown DPI with full parsing of all protocol headers and their

[2]By choice, U-Filter does not apply policies to the packets that are received and generated by the router itself, e.g., for management purposes.

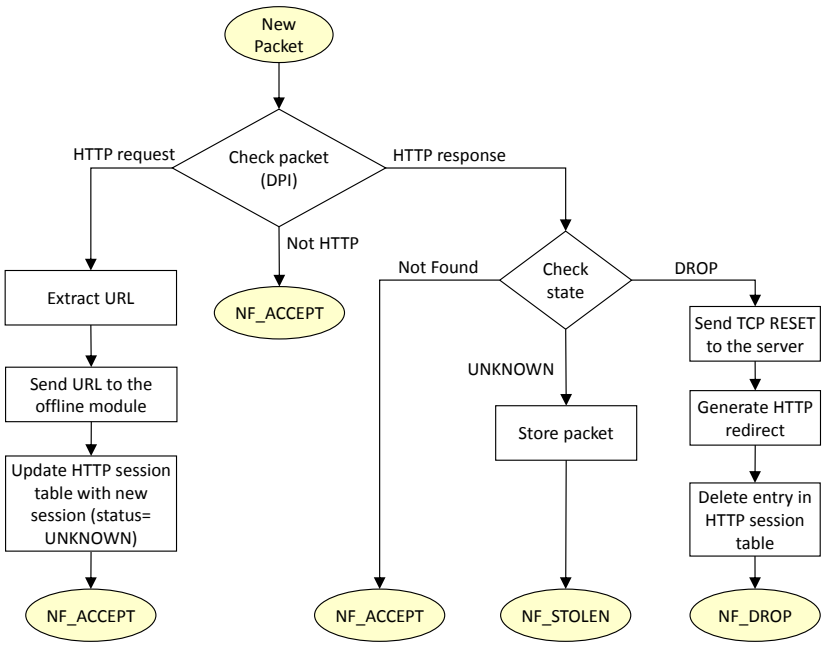


Fig. 4.7 Summarized workflow of the online module.

fields. Instead, it performs a lightweight parsing to locate the beginning of the TCP payload and a *string checking* (instead of regular expressions) just on the *initial bytes* of the payload, which is a reasonable assumption that is discussed in Section 4.3.1. In fact, our experimental validation (Section 4.4.3, Figure 4.13) confirms that the online module does not introduce noticeable overhead in the traffic processing.

In case of an HTTP request, the URL is extracted and sent to the offline module by pushing a new entry in the (shared) URL queue (Figure 4.5), which includes the TCP session identifier to later match the verdict from the policy server with the corresponding HTTP session. A new entry is also created in the HTTP session table; as shown in Figure 4.4, it includes the TCP session identifier (as a key), a session status flag that is marked as `UNKNOWN`, and an additional field that is left empty. Afterwards the packet is allowed to be forwarded by returning `NF_ACCEPT` to `netfilter`.

When an HTTP response is received, the module checks the status in the HTTP session table and acts according to the three possible scenarios:

- The lookup is successful and the requested URL is forbidden (`DROP` in the session status flag). The HTTP response is dropped (i.e., a `NF_DROP` is returned

to `netfilter`), and two new packets are generated: (*i*) a TCP RESET message sent to the web server to forcibly close the connection and (*ii*) an HTTP redirect message sent to the client in order to show the user a courtesy web page notifying that the requested web resource was blocked. Moreover the entry is removed by the HTTP session table.

- The lookup is successful but the system is still waiting for the policy server to respond (`UNKNOWN` in the session status flag). This occurs when the response from the web server arrives *before* the one from the policy server. In this case the HTTP response packet is put on hold by returning `NF_STOLEN` to `netfilter` and saved in the proper `skbuff` structure (shown in Figure 4.4) of the HTTP session table entry, waiting for the arrival of the answer from the policy server. This is the only case in which the user experiences an additional delay compared to a scenario where U-Filter is not deployed; a characterization of this delay will be provided in Section 4.3.3.

- The lookup is unsuccessful. Our algorithm interprets this condition as the URL being allowed, hence the HTTP response is forwarded to the client. Since in common URL filtering applications most URLs are not to be blocked, this design choice allows considerable space savings in the HTTP session table (Figure 4.4), as we avoid explicit entries for all the sessions that correspond to 'accepted' URLs.

Notably, the algorithm needs to hold (hence, store in the kernel session table) no more than *one* packet per HTTP session. In fact, even if other segments of the HTTP answer are in fact delivered to the destination, the TCP layer on the destination host cannot reconstruct the entire message because of the missing packet, which is the first segment of the HTTP response. This prevents the message to be actually delivered to the application (e.g., web browser) while keeping at minimum the memory storage requirements in the residential gateway. However, this solution also causes the transmission of some duplicated packets, which we analyze in Section 4.4.2 and that are discarded by U-Filter since they are equal to the packet already on hold.

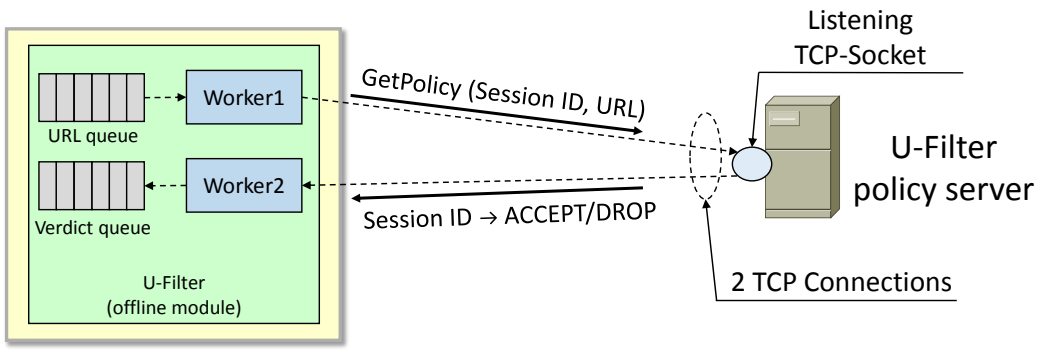


Fig. 4.8 Offline module user space process.

4.2.6 Offline module

As depicted in Figure 4.2, the offline module is split in two portions, the first one operating as a process in user space, while the other operates as a thread in kernel space. The former is in charge of the communication with the policy server, as shown in Figure 4.8, while the latter executes the workflow summarized in Figure 4.9.

The user space process retrieves URLs from the URL queue and sends them to the policy server, which provides decisions stating whether they are acceptable or to be blocked. These decisions are then pushed in the shared verdict queue, together with the same TCP session identifier that was stored in the corresponding URL queue entry.

The entries in the verdict queue are retrieved by the offline module thread in kernel space, which reads the enclosed decision. In case the resource is legitimate (the entry contains the `ACCEPT` flag), it checks whether a packet is stored in the HTTP session table entry corresponding to the TCP session key present in the verdict queue entry. This packet, if present, is injected back into the networking stack of the operating system, exactly in the same point of the `netfilter` chain where it had been stolen, so that the packet is processed by any other software relying on `netfilter` (e.g., NAT). The HTTP session table is then updated by deleting the entry since, as mentioned earlier, the absence of an entry is interpreted as an `ACCEPT` verdict. The `skbuff` structure containing the first packet of the HTTP response is stored in a memory location managed by the operating system, hence the offline module leverages the kernel space thread to access it.

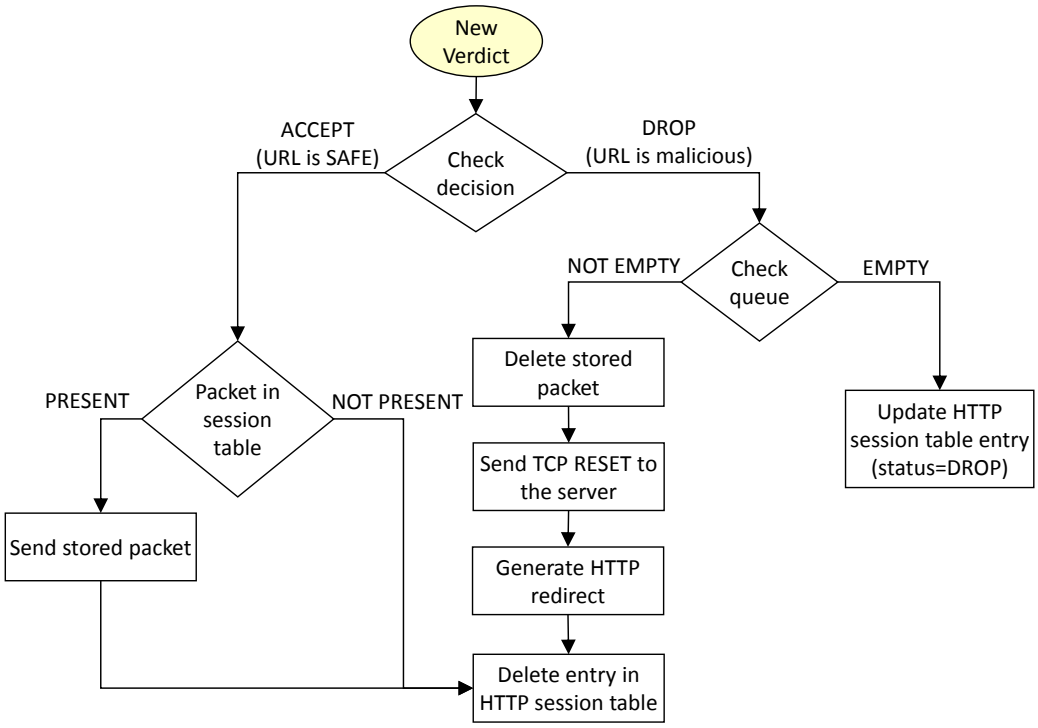


Fig. 4.9 Summarized workflow of the offline module kernel thread.

In case the resource is not legitimate (the verdict queue entry contains the `DROP` flag), if no packet is found in the HTTP session table entry, the session status flag is updated to `DROP`, thus the online module will drop the response packet when it arrives. If a packet is already stored in the HTTP session queue entry, the offline module performs the same actions previously described for the online module in case of a `DROP` policy. Additionally the packet is dropped, so that the client cannot reassemble the HTTP response.

Additionally, the last N unauthorized URLs are cached in the offline module. Each URL is first looked up in the ad-hoc verdict cache and, in case of a hit, there is no need to interact with the policy server and redirection to the courtesy web page can be immediately implemented, thus reducing the overhead for the module.

4.2.7 Communication with the policy server

The U-Filter offline module exploits two different parallel threads to interact with the policy server, each one using a distinct TCP connection as shown in Figure 4.8. The two threads establish the TCP channels when the system starts, hence enabling the offline module to send immediately a query to the policy server when needed, without the overhead (and the consequent latency) of the TCP handshake[3].

The offline module exploits these threads to implement an asynchronous communication with the policy server, separately processing the requests and the replies without any wait. The first thread cyclically collects every new entry present in the URL queue and sends the URL and the TCP session identifier to the policy server, which replies with a message on the second thread, using the second connection, containing the same Session ID and a single binary information (`ACCEPT`/`DROP`) that is used to push a new entry in the verdict queue. This solution allows to process as fast as possible both new entries in the URL queue and new replies from the policy server. The Session ID sent back and forth is used to correlate the requests with the replies, so that there is no need to share data between the two threads. Since the requests are sent sequentially, the policy server can adopt different techniques to efficiently parallelize the policy checking, such as spawning new threads without the necessity to open a dedicated TCP connection for each of them.

It is worth noting that most TCP implementations are designed to use the Nagle algorithm by default, in order to reduce the congestion of the network and increase bandwidth efficiency at the expense of latency [50]. This algorithm buffers application data until all the previously sent packets are acknowledged or the data reach the Maximum Segment Size (MSS). In this way the probability of having small packets in the network (i.e. packets smaller than the MSS) is strongly reduced, thus limiting the overhead of TCP headers, allowing for a more efficient use of transmission links and reducing the burden on routers in terms of packets per second to be processed. This behavior is particularly harmful for U-Filter, since both the offline module and the policy server always send very small packets, that most of the time would be delayed up to one RTT. It is therefore crucial that the offline module and the policy

[3]The messages sent to and received from the policy server are not intercepted by the callback of the online module, since they are addressed to the local host and do not cross the `NF_IP_FORWARD` hook, where the callback is registered.

server disable the Nagle algorithm (typically with the `TCP_NODELAY` socket option) when establishing the two connections.

4.3 Discussion

This section analyzes the proposed technique in terms of possible limitations (among the others, its applicability to encrypted traffic), and it performs a theoretical characterization of the delay that can be possibly added by U-Filter on real network traffic, which will be validated in the next section dedicated to experimental evaluation.

4.3.1 General limitations

The proposed solution has been designed with the aim of providing small delay and low overhead on resource-constrained residential gateways. This was traded for some limitations compared to more complex solutions adopting a full-stack HTTP proxy.

The matching process is meant to keep the number of string matching operations as small as possible, and surely it has to avoid to completely inspect the entire payload of all the packets in order to identify HTTP messages and extract URLs in a reasonable amount of time. Therefore, this solution does not handle correctly packets where the HTTP header is not at the beginning of a packet. This is not a relevant limitation since the problem arises only when HTTP pipelining[4] is enabled, which is rarely the case in common browsers [51, 52]. The matching algorithm also cannot handle sessions where the header of the HTTP request spans multiple packets and the necessary fields (e.g., the *Host* field) are not on the first one. According to [53], less than the 5% of HTTP requests are bigger than the common 1500 byte Ethernet maximum transmission unit. Considering that large HTTP requests are often POST messages carrying a long payload, e.g., users submitting the content of a form to a web service[5], the possibility that the URL cannot be extracted from the first packet is presumably much smaller than this amount.

[4] HTTP pipelining allows a client to send multiple HTTP requests on a single TCP connection without waiting for the corresponding responses. It requires support in both the client and the server.

[5] It is worth noting that this case falls outside the scope of U-Filter, as the apparent URL submitted in an HTTP POST request contains, in fact, user data. As a consequence, this would require a more sophisticated filtering mechanism based on a *content* inspection, not just *URL* inspection.

Moreover, various encapsulation techniques (e.g., GRE tunnels) are not supported by the presented version of the algorithm. These limitations can be avoided at the cost of additional complexity of the URL extraction procedure.

4.3.2 HTTPS

HTTPS uses data encryption to guarantee confidentiality, which makes traffic opaque to a possible observer. As a result, any in-network service requiring visibility into application layer content, such as U-Filter, becomes ineffective. Several studies [54–56] have addressed the problem of HTTPS traffic processing in middleboxes, which shows that this is a general open problem, not specific of U-Filter. As a sample general solution, [54] proposes an evolution of HTTPS that supports the operation of trusted middleboxes while retaining the security properties of HTTPS. We leave as future work the analysis of the interaction of U-Filter with such solutions.

We can envision a number of ways to enable U-Filter to operate (possibly with limited capabilities) on HTTPS traffic. A first option is to deploy a *trusted proxy* [57], such as the one presented in [58], at the cost of a significant processing overhead, which inevitably limits the performance on a resource constrained device like a residential gateway, as shown in Section 4.4.4 with respect to a similar solution.

Secondly, U-Filter can be extended to inspect unencrypted messages exchanged during the TLS session establishment, extract the domain name (from the fields Common Name, Subject Alternative Name or Server Name Indication), and enforce a policy according to the extracted value. With this solution it is possible to block only an entire domain, not just a single resource. It is worth noticing that a client can resume a previously established TLS connection with a web server by sending a past TLS session ID in the first message, which results in an abbreviated handshake without the exchange of the server domain name. Thus, if the initial connection was not inspected (e.g., because it was performed on a different, unprotected network), it is not possible to discover the server domain name by looking only at unencrypted data. Although this happens only in a quite uncommon network setup, it is to be kept in mind that the solution is not bullet proof.

As studied by [59], the cost of the security provided by HTTPS is non-negligible in particular in case of mobile devices and smart objects. In addition, there are a number of applications for which confidentiality is not strictly required, for which

their users may not willing to pay the additional cost of the encryption. Therefore a significant fraction of HTTP traffic is expected to remain unencrypted in the near future. Although we leave to future work the architectural and implementation details of a solution to support HTTPS traffic, we envision U-Filter as a low-cost solution for URL filtering on the fast path of HTTP traffic, while HTTPS traffic can be steered toward a slower path, where a trusted proxy is used to provide the same level of policy enforcement.

4.3.3 Delay characterization

In this section we analyze the additional delay introduced by U-Filter to identify the components that can be relevant and must be evaluated to quantify the impact on the user experience.

Specifically, the delay experienced by the end user when requesting a web page depends on: (*i*) the time for having a verdict from the policy server T^P, (*ii*) the time until the first packet of the response from the webserver is received T^W, (*iii*) the difference between (*i*) and (*ii*) Δ_{delay}, as detailed in Figure 4.10. The latency in the communication from the client to the residential gateway is not relevant in this context since it is not affected by the presence of U-Filter.

Let's first characterize T^P. When U-Filter receives the first packet of an HTTP request, the online module extracts the URL, pushes a new entry in the URL queue and sends the HTTP request forward. The entry spends a time T_{Uqueue} in the URL queue, until it is extracted by the offline module and sent to the policy server, with a time $T^P_{req,tx}$ required to transmit the bits on the channel. The verdict is available to the offline module after a Round-Trip Time RTT^P, a time T^P_{proc} required by the policy server to check its database and choose a verdict, and a time $T^P_{resp,tx}$ needed to transmit the response into the channel. At this point, the verdict is stored as a new entry in the verdict queue. An additional queuing time T_{Vqueue} lapses before the entry is retrieved by the offline module kernel thread and the proper action is performed to unlock the response. As a result, the total delay introduced by the policy checking process is equal to:

$$T^P = T_{Uqueue} + T^P_{req,tx} + RTT^P + T^P_{proc} + T^P_{resp,tx} + T_{Vqueue} \tag{4.1}$$

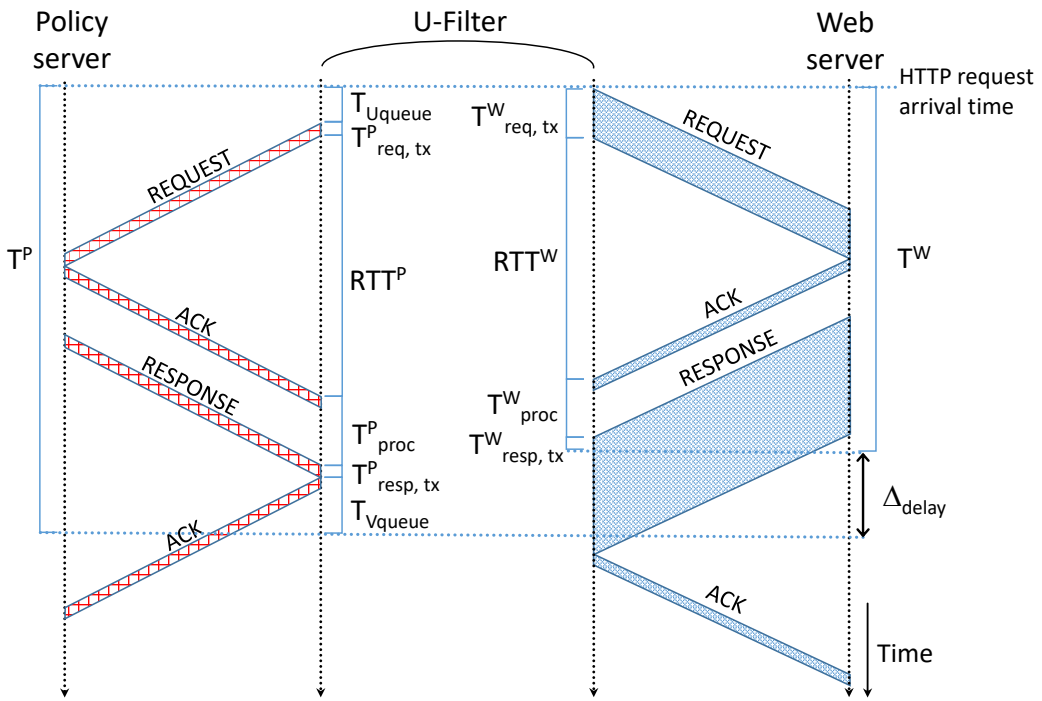


Fig. 4.10 Delay characterization.

Moving now to the characterization of T^W, the time required to receive the first packet of the HTTP response from the web server is given by:

$$T^W = T^W_{req,tx} + RTT^W + T^W_{proc} + T^W_{resp,tx} \tag{4.2}$$

where:

- $T^W_{req,tx}$ is the HTTP request transmission time;
- RTT^W is the Round-Trip Time with the web server;
- T^W_{proc} is the time taken by the web server to provide the HTTP response (fetch a file, execute server side computation, query a database, etc.);
- $T^W_{resp,tx}$ is the time needed to transmit the **first packet** of the HTTP response.

The interval:

$$\Delta_{delay} = T^P - T^W \tag{4.3}$$

when positive, is the delay that U-Filter adds to any HTTP request. Experimentally, we observed that T_{Uqueue} and T_{Vqueue} are negligible, since the two consumer tasks are rather fast. Moreover, $T^{P}_{req,tx}$ is always less than $T^{W}_{req,tx}$, since the request to the policy server contains only a small subset of the data contained in the HTTP request. Similarly, $T^{P}_{resp,tx}$ is always less than $T^{W}_{resp,tx}$, since the policy response packet is very small (it consists only of the session ID and a binary flag). Consequently, the most significant components of the U-Filter delay are the Round-Trip Times and processing times.

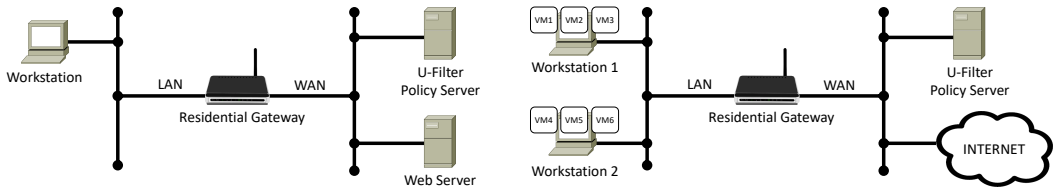
In case Δ_{delay} is negative, the user experience is completely unaffected by the presence of U-Filter. Even when Δ_{delay} is positive, though, thanks to the parallelization described in Section 4.2.7, the overall delay in a web page load time is not noticeable if the distance and the processing time of the policy server T^{P}_{proc} are comparable with the ones of common web servers, as shown in Section 4.4.

4.4 Experimental validation

In order to validate the proposed solution we conducted a broad range of experiments. Specifically our goal has been to study the interaction between the presented algorithm and TCP, as well as the conditions in which a web page load time is increased, quantifying to what extent the user experience is affected.

4.4.1 Testbed setup

We deployed U-Filter on a commercial low-cost residential gateway, a TP-Link Archer C7 (single core MIPS32 CPU clocked at 720MHz, 16MB Flash, 128MB RAM) running *OpenWrt* 12.09 [60] with the version 3.3 of the Linux kernel. OpenWrt is an open source operating system specifically optimized for the execution on resource constrained residential gateways. As shown in Figure 4.11, multiple workstations (whose number and setup varies according to the specific test) acting as clients are connected on a Gigabit Ethernet LAN representing the “domestic side” of the residential gateway. Another 1 Gbps interface (“WAN side”) hosts the policy server and the traffic sink of our experiments, which is represented by a web server during TCP interaction and throughput experiments or a vanilla Internet connectivity when evaluating browsing experience. All the workstations and the servers are



(a) Testbed to analyze the interaction with TCP and to evaluate the maximum throughput.

(b) Testbed to evaluate the browsing experience.

Fig. 4.11 Testbed setup.

equipped with an Intel Core i7-4770 CPU and 32GB of main memory in order to guarantee not to become the bottleneck.

Since a production-grade policy server is not in the scope of this work, we use a policy server that gives always a positive verdict, with a customizable delay in order to simulate the processing time. Moreover, in the policy server we use *Linux Traffic Control* (`tc`) to add a custom delay to any outgoing packet in order to simulate various network RTTs.

To generate single HTTP requests we use `curl` and `ab` [61], while for real-life simulations we start multiple VMs on the workstations to emulate multiple end-users. Each VM runs an instance of *WebTrafficGenerator*[6], an automation tool that can drive a web browser to replay a user browsing history. For every entry in the provided browsing history, the browser loads a complete web page (i.e. retrieving the web page with all the associated resources such as images, javascript files, etc.)[7]. In this respect, *WebTrafficGenerator* can also issue HTTPS requests, which happens when a page, appearing in HTTP in the browsing history, includes content that has to be retrieved using an encrypted connection. The time between multiple web page requests, a.k.a. the *Thinking Time*, is randomly selected using a random variable with the same statistical distribution as the actual thinking time of the user as measured from his/her browsing history. A realistic thinking time is required not only to simulate a real user behavior, but also to avoid that web services (e.g. Google) recognize that the client is an automaton and thus provide a different response web

[6]https://github.com/netgroup-polito/WebTrafficGenerator

[7]The community have not yet reached a consensus on when a web page should be considered completely loaded. Particularly, *WebTrafficGenerator* considers a page complete when the javascript "`onload`" event is fired on the "`body`" HTML tag.

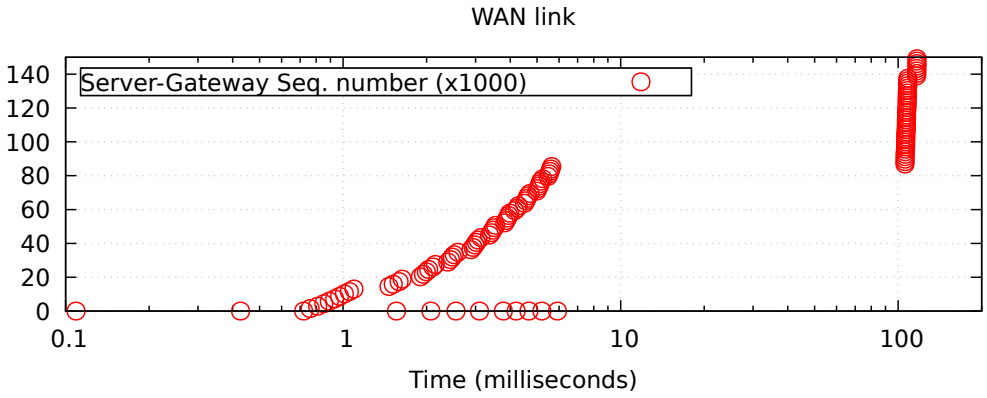
page with the intent of testing whether or not the user is human. In the event that a new request must start before the previous web page is completely loaded, the tool creates a different browser window, in order to load multiple web pages in parallel (which simulates multi-tabbing).

4.4.2 Interaction with TCP

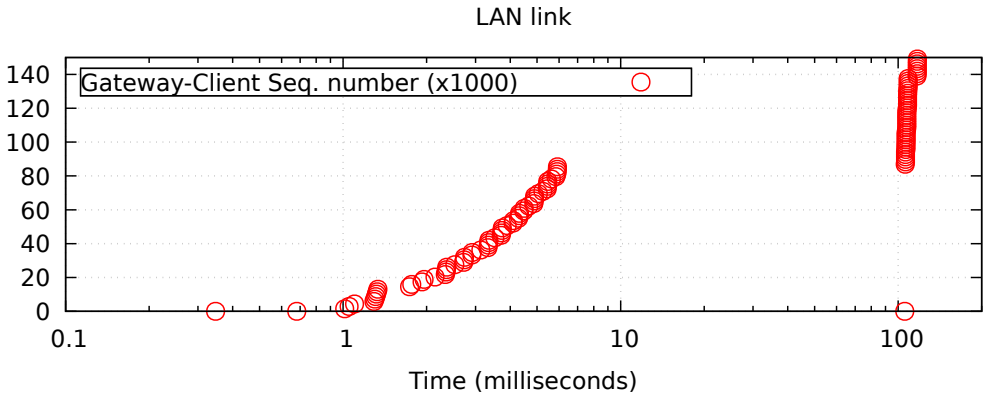
This section shows how the TCP algorithm reacts when one specific packet (the first packet of an HTTP response) is repeatedly lost on its way to the destination, for a certain amount of time. The aim of this analysis is to show that U-Filter has been designed taking into mind the peculiar characteristics of the TCP protocol, hence our algorithm that possibly delays the first packet of the HTTP response does not cause additional delay in the TCP data exchange.

To reduce external interferences, in this test we use a web server directly connected to the WAN interface of the gateway (as shown in Figure 4.11a) running the *Apache HTTP Server 2.4.7*; T^W measured in this setup is less than 1 ms, thus we can consider $\Delta_{delay} = T^P$. Moreover, in this test `tc` is disabled in the policy server, hence the RTT is negligible and we can consider $T^P = T^P_{proc}$. A client workstation runs `curl` to request a 512 KB web page stored on the webserver. The gateway executes U-Filter with a fixed $T^P_{proc} \approx 100$ ms delay in the policy server response. As detailed in Section 4.2.5 and 4.2.6, only the first packet of any HTTP response is buffered by U-Filter. In the scenario created for these experiments, such packet is eventually forwarded to the client about 100 ms after the HTTP `GET` request traverses the residential gateway. All subsequent packets are forwarded correctly. We capture the traffic on both the LAN and WAN links of the residential gateway and extract the sequence numbers (SEQ) of the TCP segments from the web server to the client and the acknowledgment numbers (ACK) of the ones from the client to the webserver, together with their timestamp. The resulting data are presented in Figure 4.12 (the SEQ and ACK numbers are relative).

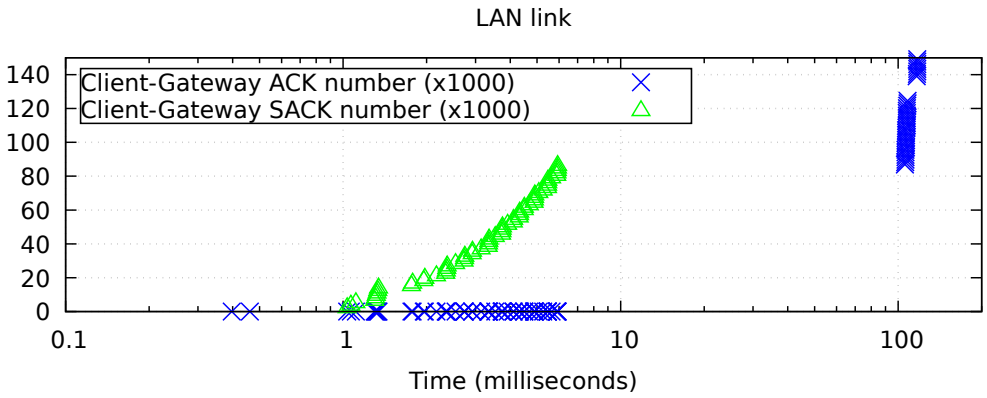
This experiment enables us to observe how a TCP connection progresses during the U-Filter operation. The presented results show that, while the first TCP segment of the HTTP response is blocked, the server TCP endpoint sends the subsequent segments as well as duplicates of the first segment (visible only on the WAN side, in Figure 4.12a), until the TCP window is full. As expected, the TCP receiver repeatedly



(a) Response packets timing on the WAN link



(b) Response packets timing on the LAN link



(c) Acknowledgement packets timing on the LAN link

Fig. 4.12 Progress of a TCP session.

acknowledges the segment arrived before the one missing (Figure 4.12c); specifically one ACK is sent for each of the subsequent segments received out of sequence. All the modern TCP implementations include the *TCP selective acknowledgment (SACK) option* [62] in the duplicated ACK, which is used to selectively acknowledge correctly received segments logically following the missing one(s). Thanks to the selective acknowledgments, these segments are not re-transmitted, as it happens for the blocked segment, as the traditional Go-Back-N algorithm would require. When the blocked packet is released (after 100 ms in our experiment, as shown in Figure 4.12b) and properly delivered, all the previously received segments are cumulatively acknowledged and the transmission can continue from a new segment (Figure 4.12c).

Abiding by *TCP Fast retransmit* [63] algorithm, the web server re-sends the blocked segment for every 3 duplicated acknowledgments. These re-transmitted segments are the only overhead induced by U-Filter. In our test these duplicates amount to 12.8% of the packets sent by the server during Δ_{delay}, and half that number if we consider *all* the packets transmitted during the same interval; however, considering the entire lifespan of the TCP connection, this overhead accounts (in average) no more than 1.6% of all the packets, which can be considered negligible.

From the point of view of the users' experience, selective acknowledgments are particularly beneficial because, even if the policy server replies after the web server (i.e. Δ_{delay} is positive), the actual delay perceived by the user is smaller than Δ_{delay} because several TCP segments are correctly received during the Δ_{delay} interval and are ready to be used to render the web page as soon as the missing segment is delivered.

4.4.3 Browsing experience

This section presents the results of several tests executed in a realistic scenario to show how much a real user browsing experience is affected by U-Filter. Using the testbed in Figure 4.11b, we launched *WebTrafficGenerator* in 6 VMs (running on 2 workstations) in order to simulate 6 users simultaneously browsing the Internet. This number of concurrent users is reasonable for a residential gateway. Moreover, with a large number of users, the browsing experience would be limited by the network

Table 4.1 Inferred RTT values with the policy server in different locations (RTT^P).

Location	Type of measure	RTT
POP	Median	25 ms
	90^{th} percentile	100 ms
Data Center (DC)	Median	45 ms
	90^{th} percentile	200 ms

speed. As expected, the latency of the policy server proved to be the parameter that has the greater impact on the user-perceived performance of U-Filter.

In every test, a single VM browses 600 web pages collected from the browsing histories of 30 anonymous users (we consider only web pages downloaded using HTTP, since those using HTTPS are irrelevant for U-Filter). In order to use realistic values for the policy server processing time and RTT, we analyzed several traffic traces captured using Tstat [64] during 24 hours in 4 different points of presence (POPs) of an Internet Service Provider (ISP) and extracted the median and 90^{th} percentile values for the RTT of HTTP requests and processing time of web servers. Tstat infers the RTT from the POP to an endpoint by measuring the inter-arrival time of a packet and its acknowledgment and infers a web server processing time by measuring the interval between the arrival of the acknowledgment for the request and the arrival of the first response packet. In fact, a host's operating system usually sends a TCP ACK as soon as a packet is received.

Table 4.1 shows the statistical values for the RTTs from a client to the POP and from a client to the destination server, supposedly in a data center (DC). We use these values in our tests to simulate the RTT in the case that the policy server is either in the POP or in a remote data center. Additionally Table 4.2 shows the statistical values of the processing time for web servers. These values are used to simulate the processing time of the policy server: since the operations performed are somewhat similar (parsing of a request, look up in a database, preparation of a response), we assume the complexity to be comparable with (or even lower than) the one of any web server.

At the end of a test, *WebTrafficGenerator* provides a file containing a summary of various aspects of every request. Among the provided values, we are interested

Table 4.2 Inferred policy server latency values (T^P_{proc}).

Type of measure	Latency
Median	2 ms
90th percentile	80 ms

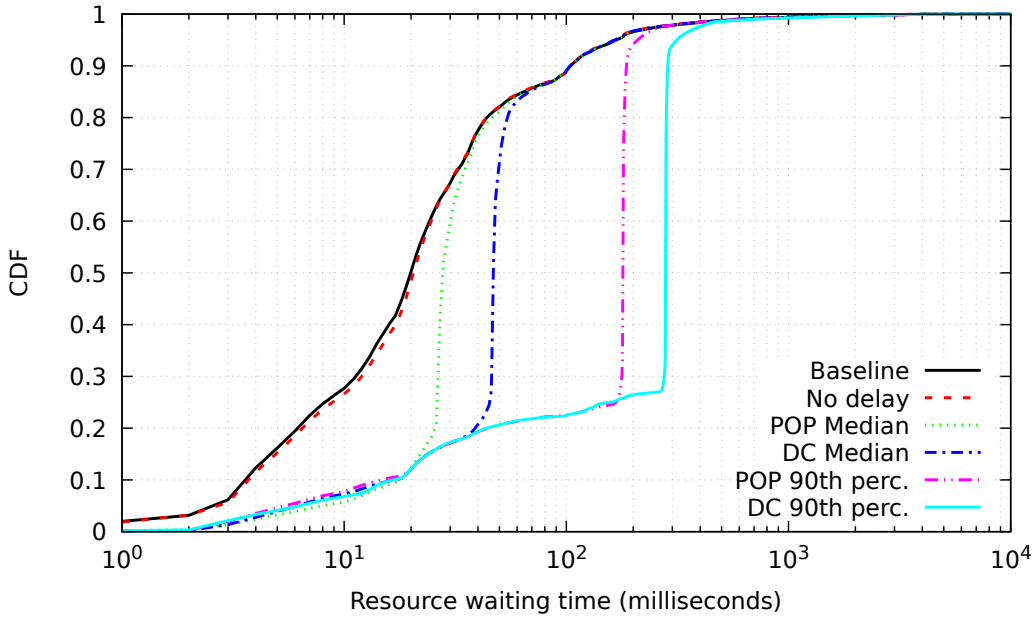


Fig. 4.13 Waiting time for a single HTTP resource - Cumulative distribution function.

in the **complete page** load time (the time needed to load the web page with all its resources, such as pictures, libraries, etc.) and the timings of the **individual HTTP requests** issued to get the main HTML page and the associated resources.

Individual HTTP requests

The timing of an HTTP request is the sum of multiple components, such as the queuing time, the DNS resolution time, the connection setup time, etc. The only component that can be affected by U-Filter is the time spent waiting for a response from the server (*waiting time*), equal to $\max\{T^P, T^W\}$, if the RTT between the client and the gateway is negligible. Figure 4.13 shows the cumulative distribution of the waiting time for HTTP requests with different values of RTT and processing time

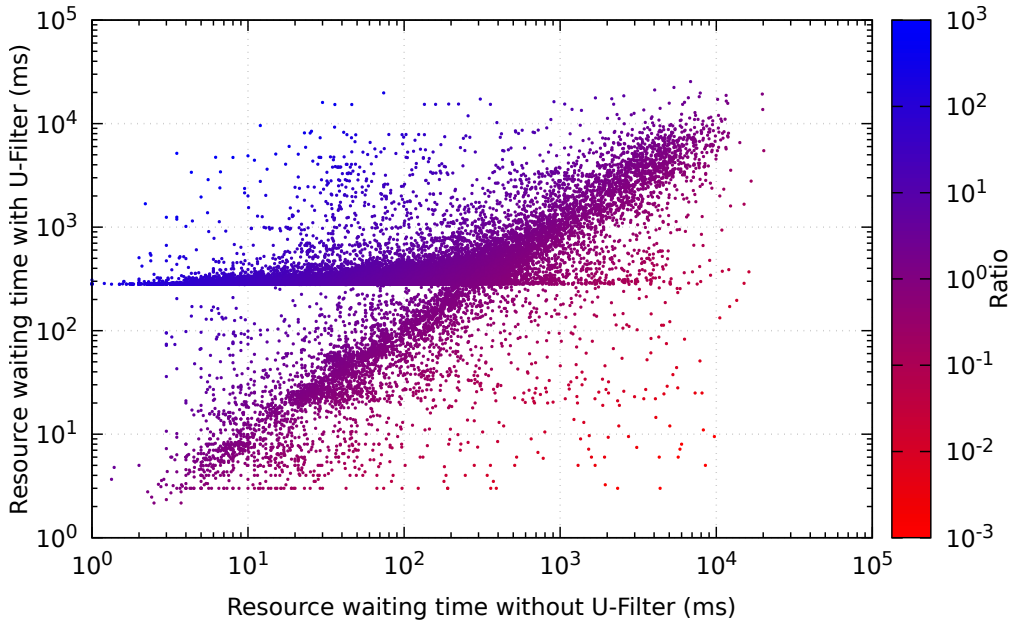


Fig. 4.14 Resource waiting time considering the 90th percentile of the processing time and RTT with the policy server in a data center.

(latency) for the policy server, together with the baseline (i.e., the latency without U-Filter) and the case in which the policy server immediately provides verdicts (in which case the delay T^P is negligible), as if U-Filter and the policy server are on the same LAN.

These results show that U-Filter adds a negligible delay if the policy server provides an immediate response, therefore proving our claim that the online module does not introduce noticeable overhead in the traffic processing. On the other hand, when the policy server response is received after a certain amount of time, the cumulative distribution is shifted toward that value, since all the HTTP responses that arrived earlier are delayed by U-Filter. In summary, the impact of U-Filter on the single resource loading time is highly dependent on the distance from the policy server and its processing time.

Considering only the worst case (i.e., the 90th percentile of the processing time and RTT with the policy server in a data center), we show in Figure 4.14 the waiting time for each requested HTTP resource, with and without U-Filter. The figure shows a cluster of requests on the horizontal line corresponding to the delay T^P, supporting the conclusion that this delay highly influences the loading time of a single resources.

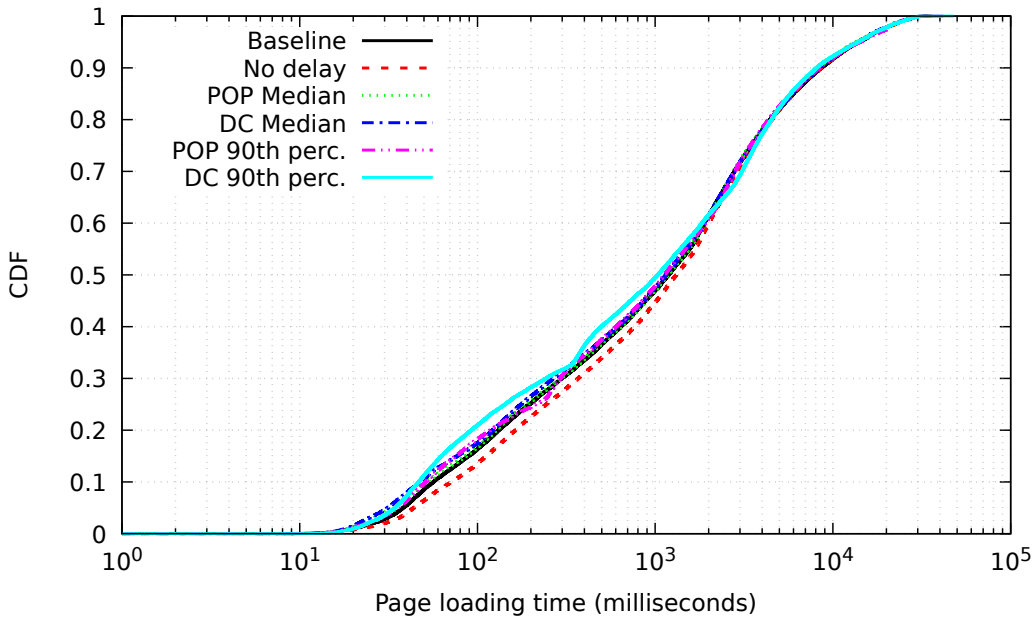


Fig. 4.15 Complete page loading time cumulative distribution.

Both figures show that, even with U-Filter, some resources are received before the policy server delay ($T^P \approx RTT^P + T^P_{proc}$). This happens because some resources are retrieved through HTTPS, even if the main HTML page is on HTTP, therefore they do not experience the policy server delay.

Complete pages

Figure 4.15 shows the cumulative distribution function of the complete web page load time, while Figure 4.16 shows for every requested URL the relation between the complete page loading time with and without U-Filter, in the worst conditions (policy server in the data center, 90$^{\text{th}}$ percentile values for RTT and latency). These results show that the impact caused by the presence of U-Filter is not noticeable, therefore we can assert that the overall page loading time is not affected by U-Filter and also the browsing experience is unaltered.

This is justified by the fact that multiple resources are requested **in parallel** by the browsers, hence the policy server processes all the requests concurrently. As a result, the increase in the overall time for loading the complete web page is not dependent on the number of resources and is, in any case, approximately equal to a single policy server delay T^P. Since the time needed to receive, parse and render the

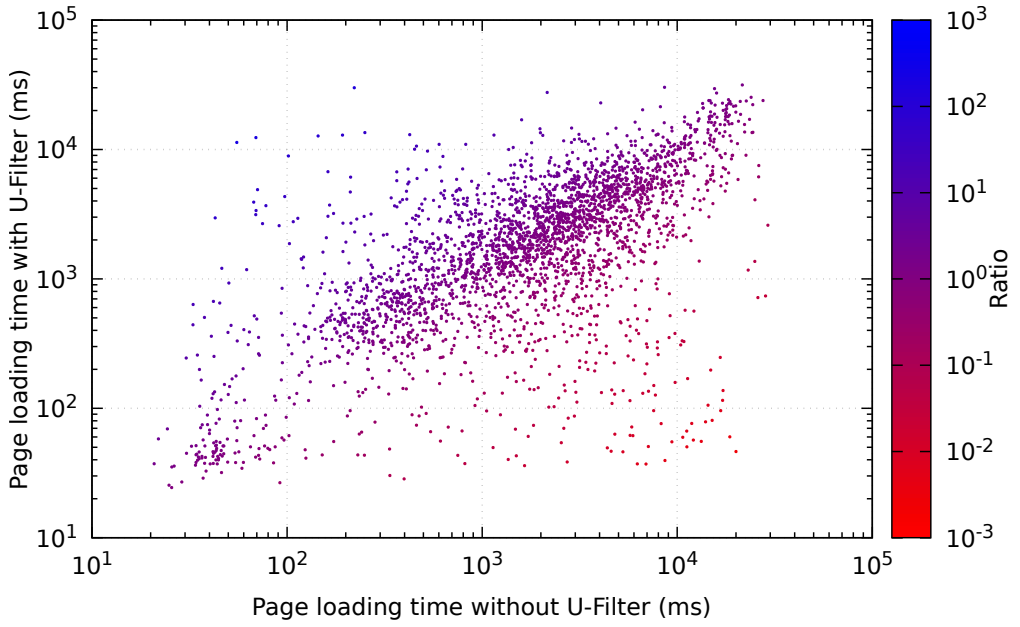


Fig. 4.16 Complete page loading time considering the 90th percentile policy server processing time with the policy server in a data center.

main HTML web page and all its resources is usually an order of magnitude greater than the policy server delay, the added latency (and the impact of U-Filter on the browsing experience) is in effect negligible.

4.4.4 Residential gateway aggregated throughput

In this section we evaluate the overhead introduced by U-Filter by comparing the average aggregated throughput of the residential gateway in 3 scenarios: (*i*) without a URL filtering service in place, (*ii*) with U-Filter and (*iii*) with *Tinyproxy* [65], a URL filtering solution for OpenWrt based on a lightweight HTTP proxy that intercepts and analyzes all the outgoing web traffic and can operate in either explicit or transparent (a.k.a. man-in-the-middle) mode. These experiments assess the impact of U-Filter with respect to the maximum forwarding capabilities of the residential gateway, which is basically limited by the CPU consumption of the on-board software.

These experiments employ the testbed setup depicted in Figure 4.11a; the policy server is configured to simulate a deployment in a data center with the median processing time and RTT, while the web server has the same RTT. The client workstation uses `ab` to request files of different sizes from the web server; each file is requested

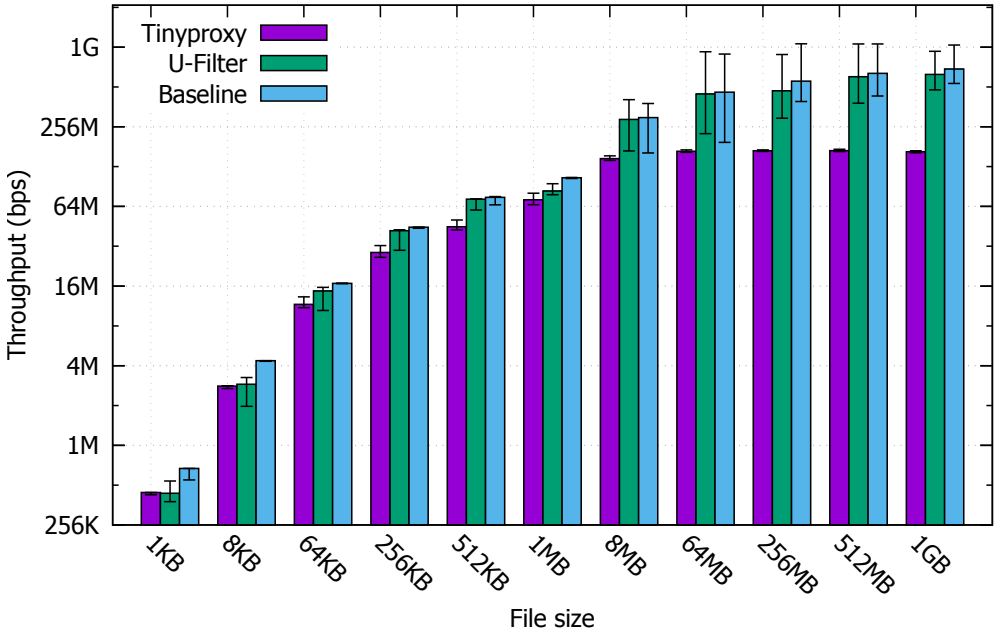


Fig. 4.17 Application-level throughput when downloading files of different sizes.

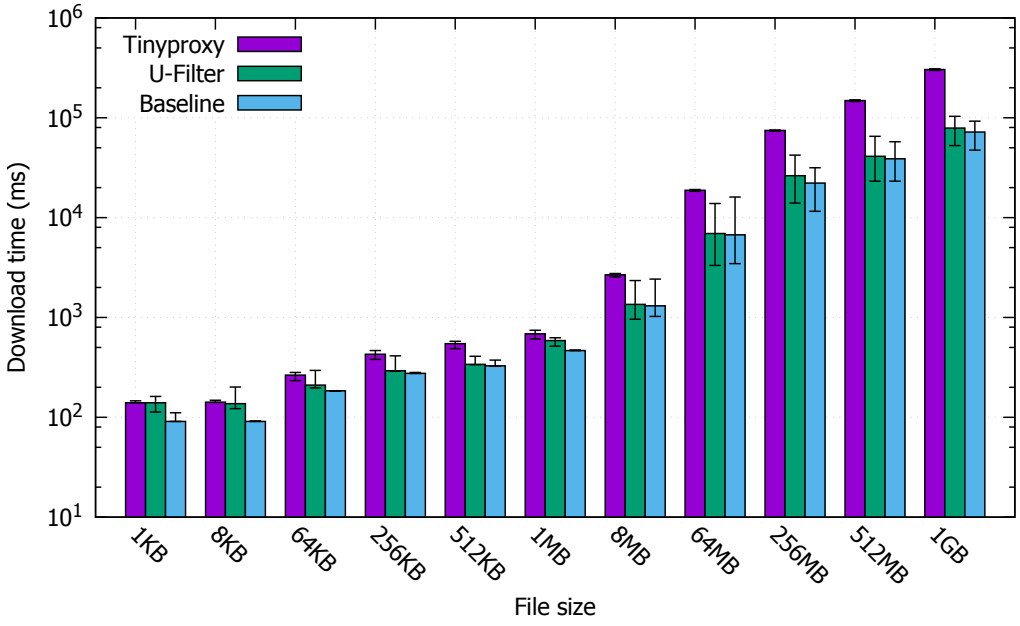


Fig. 4.18 Download time when requesting files of different sizes.

100 times. As suggested by the HTTP/1.1 standard [66] with respect to persistent HTTP connections, each client issues two concurrent requests toward the server. The goal of this experiment is to evaluate how much packet inspection and policy checking in the residential gateway affects the download speed and the latency. We show in Figure 4.17 the minimum, maximum and average application-level throughput for the 3 scenarios, while in Figure 4.18 we show the time needed to download the entire file.

These results show that the throughput and the download speed reached with U-Filter are higher than with Tinyproxy for files larger than 8 KB, while for small files the two solutions show the same level of performance. In fact, with very small files, we experience an additional small delay with U-Filter, compared to the baseline. We ascribe this delay to the time needed for the context switch between the online and offline module, given that the residential gateway has a single core. This delay is negligible for larger files, for which U-Filter provides almost the same performance reached without the filtering service in place. We expect that a residential gateway with at least a dual core processor would not experience this delay, therefore U-Filter would provide the same level of performance as the baseline. However, even with a single core gateway, the impact of U-Filter on the download time is only 3% with large files and never exceeds 54%, while Tinyproxy has an overhead ranging from 44% to a remarkable 322%. As an example, the download of a 1 GB file requires approximately 1 minute and 12 seconds without a filtering service, 6 seconds longer with U-Filter and more than 5 minutes with Tinyproxy.

It is worth mentioning that U-Filter can easily implement a whitelist containing the addresses of trusted devices or applications whose traffic should not be filtered. This is a useful feature that allows to avoid the additional delay for delay-sensitive clients. Similarly, the user can define an explicit blacklist, listing the type of traffic that should be immediately blocked.

4.4.5 Memory footprint

Given the limitations in terms of available memory in current residential gateways, we extracted the number of pending entries in the HTTP session table every time a new HTTP request was received and plotted the resulting probability distribution in Figure 4.19 in order to assess the impact of U-Filter in terms of memory consumption.

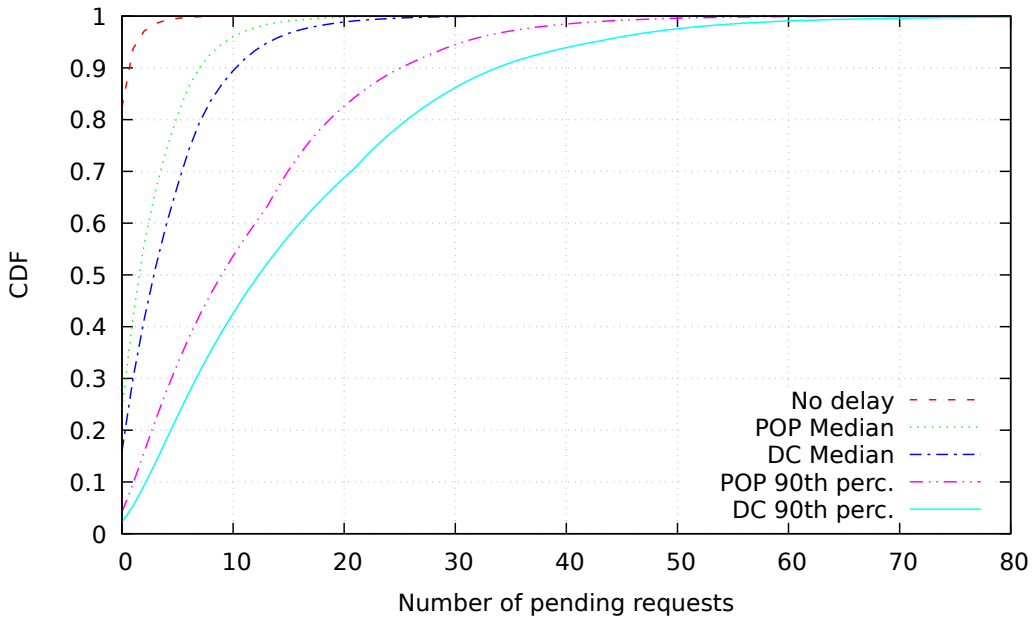


Fig. 4.19 U-Filter load.

Clearly, the memory footprint is proportional to the rate of issued requests and the measurements made in the realistic scenario confirm the small memory footprint of U-Filter: even in the worst case, the number of pending entries are always less than a hundred. In the case in which every entry stores a packet (usually 1518 bytes at most), together with IP addresses (8 bytes), TCP ports (4 bytes) and a binary session flag, the HTTP session table requires less than 200 KB of main memory, a value far below the memory size of low-end residential gateways (usually in the order of at least tens of MB).

4.5 Related work

Currently several solutions for filtering traffic based on URLs are available commercially or as open source packages, often used as parental control or ad block. Many are based on software executing on the client machine to control outgoing traffic. Among them, it is worth mentioning *k9 Web Protection* [67], a powerful free software for URL filtering that comes with a large database of URL categorization data. New websites are categorized in real-time and their information published on a

server that is used to update the local database. This software needs to be installed on any device that must be protected and is tuned to run on common PC hardware.

Among existing parental control solutions that do not require execution of a software agent on clients, some are based on applying the filtering policing in the DNS server [68]. While this is a low complexity and efficient solution that enables achieving high performance, it is not effective as it can be easily bypassed choosing a different DNS server. Moreover, filtering is based on server domain names rather than URLs, as required when the same server or name domain can deliver both appropriate and inappropriate content, such as in case of public services like `facebook.com`.

As an alternative approach, filtering policies can be applied by network appliances on the path of the protected client traffic. *Blue Coat WebFilter* [69] is a sophisticated URL filtering solution that runs on business level network appliances and provides policy enforcement on web traffic, blocking malware downloads and web threats. *WebFilter* combines URL filtering and anti-malware technologies, exploiting an engine with a local rule database continuously updated from a remote master database. The engine detects hidden malware and provides reputation and web content categorization based on input from actual users.

None of the above-mentioned solutions is designed to run on resource-constrained devices, such as a typical residential gateway, which would not ensure acceptable performance when executing computationally intense tasks. Among the efforts to integrate web filtering service in low-end residential gateways, the ones related to the OpenWrt platform are noteworthy, such as Tinyproxy [65]. Tinyproxy can filter HTTP requests checking their URL against a list of regular expressions contained in a local file, which may be rather big and needs to be frequently updated. A similar technology has been proposed in [58], where an access gateway performs mobile app policy enforcement deploying a transparent HTTPS proxy to gain access to *encrypted* traffic, extract relevant field values, and pass them to an external policy-checking module. However, deployment of an HTTP proxy is critical on resource-constrained devices since it must terminate all the TCP connections, pair them with new TCP connections with the remote endpoint, parse every packet, identify and extract patterns of interest, and match them against a large blacklist. Therefore it becomes easily a bottleneck with high traffic loads, thus impacting user experience.

The work presented in [70] represents an attempt to perform efficient HTTP traffic filtering in *OpenWrt*. The authors propose a two-tier architecture, with a

kernel module that intercepts and analyzes HTTP traffic and a user-space process in charge of policy compliance checking. The computational load of the user space module, that performs string matching on URLs, grows with the length of the list of rules, and so does the introduced delay. Consequently, when this approach is implemented on a residential gateway with limited resources, only short lists can be supported without user experience degradation, thus limiting the effectiveness of the policy enforcement system. Moreover, the proposed architecture makes it difficult for a trusted third-party to push real-time updates to the local database in order to ensure prompt detection of newly discovered threats. Finally, the URL analysis is performed by each edge systems in isolation, hence excluding the possibility of a (centralized) cross-correlation mechanism that identifies new threats by analyzing URLs requested from different sources.

Traffic processing in residential gateways has been proposed also in the context of Network Function Virtualization (NFV) [33, 71]. An existing NFV infrastructure can employ residential gateways to deploy lightweight Native Network Functions [11] or eBPF data plane programs [34], in order to provide delay-sensitive services to the user, while computation intensive services are hosted in the data center of the service operator. This solution offers flexibility in the type and number of network services that can be provided and represents an interesting target platform for the deployment of U-Filter.

4.6 Conclusions

This chapter presents U-Filter, a distributed system for efficient HTTP traffic filtering in resource-constrained residential gateways. Leveraging an external policy server and an intelligent combination of kernel and user space processing (and a careful implementation), U-Filter is able to inspect the URL in every HTTP request and block unwanted web pages with a very small memory footprint and processing overhead. This makes U-Filter appropriate for the deployment on resource-constrained devices and also reduces at a minimum the additional delay introduced on page download, which leaves the overall browsing experience of the user practically unaltered.

Since U-Filter operates on a packet-by-packet basis, it assumes that the entire HTTP header is on the same packet. This makes URL extraction easier and avoids to have to store additional information to correlate subsequent packets. Since the

maximum size of an IP packet is usually 1500 bytes, this does not represent a problem in a real scenario, as confirmed by [53].

The policy server, where multiple mechanisms and optimizations can be implemented, was purposely kept outside of the scope of this work as it involves a completely different set of challenges and solutions. Similarly, we did not address how providing additional information to the residential gateway can increase its efficiency in caching verdicts, thus reducing the number of interrogations. The study of such improvements is left to future work.

Future research could also study how this technique can be applied to enforce different security properties. U-Filter can extract an appropriate fingerprint from the first packet of a flow and forward it to the policy server for further analysis in order to block possible malware or Denial-Of-Service traffic, or curb data exfiltration.

Chapter 5

Packet processing in the core: a Massively Distributed Network Data Caching Platform

5.1 Introduction

Network Service Providers are usually forced to deploy multiple middleboxes in various locations of their network in order to obtain a comprehensive perspective of traffic and activities behind it. These devices monitor packets *independently* of each other, thus leading to redundant processing, duplicated reports and inefficient use of resources, which are required in large amounts given the sheer volume of traffic in today's broadband networks. Moreover, when an overall view of the global network and correlation of distributed events are required, these devices must forward captured traffic to a Network Operations Center (NOC) for a complete analysis, which significantly increases the amount of traffic in the network and leads to high resource requirements for the NOC. The situation is exacerbated by the fact that multiple copies of the same packet are captured and sent to the NOC where resource intensive de-duplication should be performed. The cost of eliminating duplicates is so hefty [73, 74] that some network administrators explicitly choose to skip it and accept the error that duplicates introduce in statistics and analytics.

The content of this chapter has been published in [72].

To address the above issues and improve the efficiency of network-wide traffic monitoring, we propose MEDINA, a highly distributed and decentralized traffic capture and processing platform. MEDINA significantly reduces the storage and processing requirements at the NOC and traffic overhead by capturing, pre-processing, and storing raw packet data directly on the packet routing nodes *themselves*. The aim of MEDINA is to take advantage of a recent trend followed by networking hardware manufactures towards systems combining networking, computing and storage [75], to enhance traffic forwarding devices with the capability to capture and process packets along a path in the network. MEDINA proposes a limited overhead *coordination and self-adaptation algorithm* to distribute tasks across multiple devices. Using such algorithm, nodes converge to a shared load distribution plan such that each packet is always captured precisely n times (where n is a parameter of the algorithm) by different nodes along the route to its destination, which ensures complete visibility on the traffic as well as fault tolerance. Different packets are processed by a different set of n nodes, which ensures distribution of the load among network nodes. The algorithm also automatically adapts to the changing traffic characteristics, thus the shared distribution plan is always aligned with the actual load of each node. Capturing traffic n times ensures that each packet is processed with the required level of redundancy: the results of the processing are stored on multiple nodes and are available even if some nodes fail or become unavailable.

Furthermore, the proposed algorithm can distribute packet capture among MEDINA nodes with custom granularity controlled by specific parameters of the distribution algorithm. For example, packets of different flows, or even chunks of individual packets, can be captured by separate nodes. Capturing chunks instead of full packets guarantees that in the event of an attacker taking control of one (or more) MEDINA network devices, he/she will not be able to gain access to the full content of the traffic, unless a large number of devices is compromised.

By applying MEDINA, packets or the outcome of their analysis do not need to be transferred to a NOC or the cloud, which avoids large additional traffic that can affect the network operation. The results of packet processing are instead stored locally on the network node(s) that captured them. Network nodes provide a push/pull interface to offer direct and efficient access to the data based on a configurable search key that can be associated to the corresponding metadata extracted from protocol headers. Such metadata can be extracted as part of the capturing process and exported to a NOC to serve as a basis to execute queries and analytics; once packets of interest

are identified as the outcome of such analysis, they can be retrieved, possibly during periods of low network utilization, through the direct access interface using the associated search key.

MEDINA nodes can implement data reduction strategies to keep the full packet data only for the a given amount of time (directly based on the available storage resources) and discard it when outdated. Ideally, this amount of time will be long enough to enable identifying an anomaly and properly counteracting it.

Existing centralized approaches [76, 77] for node coordination rely on a controller that, besides being a single point of failure, requires a large amount of resources to cope with large topologies. Moreover, such approaches limit the capacity of the system to rapidly react to unexpected traffic variations. In fact, to adapt the load distribution to a new traffic condition, the controller must gather load profiles from all the devices in the network (which also results in a large amount of traffic in the vicinity of the controller) and only afterwards can compute the optimal distribution and send the new configuration to the nodes. The additional delay, due to the round trip time with the controller, can cause a late reaction. MEDINA does not experience such delay since the approach is decentralized, where a subset of nodes cooperate to rapidly converge to a new load distribution.

The main challenges we faced in building MEDINA is the design of a decentralized coordination algorithm with minimum need to add information within packets and low control traffic overhead. We addressed this challenge by using an hash based selection mechanism and leveraging path awareness achieved through routing protocols (anyway executed by routers) or, when not possible, ad-hoc messages. Moreover, the proposed solution does not require significant changes to the data plane. In fact, the coordination is performed within the control plane and traffic selection is enforced by pushing rules in match-action tables, commonly deployed by traditional data plane hardware.

Although the MEDINA coordination and self-adaptation algorithm is presented here in the context of packet capture and processing, it offers a general solution for distributing across multiple (virtual) processors the execution of a task that operates on data flowing through a network of nodes with processing and storage capabilities. Such scenario is typical of several paradigms currently considered as having very high potential, such as Fog Computing [78], microservice architectures and the Internet of Things [79].

The contributions of this chapter are: (*i*) the design of a decentralized solution for traffic acquisition that fairly distributes the load among multiple network devices, while avoiding duplicated capture, and (*ii*) its validation through experiments and simulations. In Section 5.2 we explain our general approach and then, in Section 5.3, report on a set of experiments in a realistic scenario to validate it. Section 5.4 compares the presented solution with existing work. Finally, Section 5.5 draws conclusions and outlines future work.

5.2 MEDINA Design

MEDINA implements network-wide coordination to ensure that all the traffic in the network is captured, eliminating redundant acquisition (or limiting it to a specified replication factor) of the same packets in multiple nodes. Other approaches to network-wide coordination demand either an external static configuration [80] or a continuous communication between network nodes and a centralized entity that determines the subset of the traffic each node is responsible for [81]. Instead, MEDINA deploys a novel, *fully distributed*, **traffic assignment algorithm** that enables each node to autonomously[1] determine the subset of network traffic it will be responsible for. The algorithm leverages routing information collected by each node in support of their regular forwarding operation to achieve implicit coordination among nodes, *i.e.*, ensure that subsets of different nodes are mutually exclusive and that the union of all subsets includes all network traffic. When a MEDINA node forwards a packet, it uses a *hash-based selection mechanism* to determine whether the packet belongs to its subset of traffic. The hash based approach allows different nodes to autonomously and coherently select a subset of the network traffic. Moreover, hash-based approaches to packet processing are known to be efficient, scalable and suitable to hardware implementation, which is key in enabling wire speed operation [82].

Each component of the solution is presented in the reminder of this section after an overview of how MEDINA can be deployed.

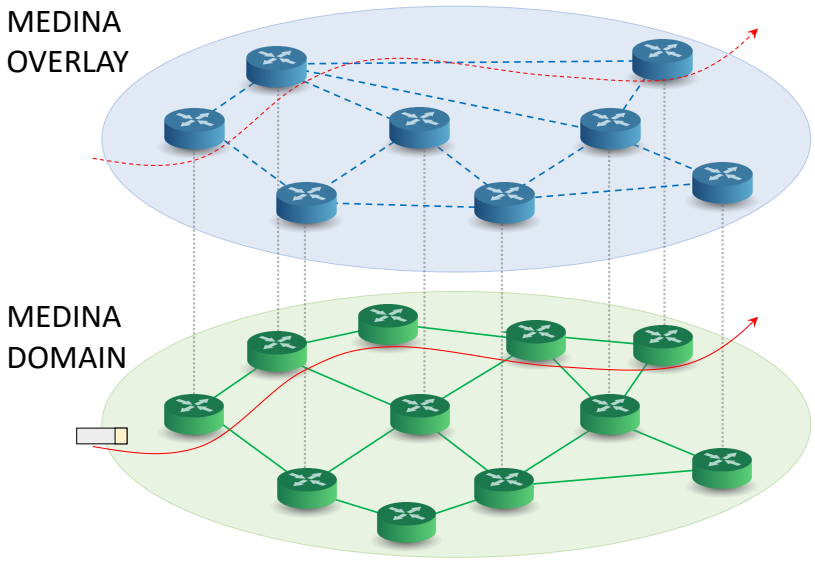


Fig. 5.1 MEDINA overlay.

5.2.1 Deployment model

The solution presented in this chapter does not require all routers in the network to be MEDINA-enabled: as shown in Figure 5.1, non-adjacent MEDINA nodes in a MEDINA *domain* form a MEDINA *overlay* working together to capture the traffic in a coordinated fashion. Hence, the solution can be deployed in an incremental way enjoying the corresponding benefits after just a fraction of network nodes have been upgraded, scaling up the capabilities of the infrastructure as the MEDINA domain becomes larger and MEDINA-enabled nodes denser. A larger number of MEDINA-enabled nodes allows to capture more traffic and/or reduce the load on all the MEDINA nodes. How the size of the deployment affects the system performance is an interesting topic that we leave for future work.

Note that while the MEDINA overlay is used for coordination and only nodes in the MEDINA overlay capture, process and store packets, the data plane is unchanged with packets being forwarded according to the physical topology. MEDINA nodes leverage topological knowledge to distribute traffic acquisition among themselves. They acquire information on the network topology and the paths that packets follow in the network by either leveraging the existing routing information base (if individual nodes operate as regular IP routers, a routing protocol based on the link state

[1]Communication among nodes is limited to the discovery phase and, optionally, the occasional exchange of constraints to adapt to variable traffic characteristics.

algorithm, such as OSPF and IS-IS, is commonly deployed) or by exchanging ad-hoc messages[2]. Starting from this knowledge, MEDINA nodes infer the path of packets through the MEDINA overlay, on which packet assignment depends.

MEDINA nodes discover the overlay, i.e., which other nodes support MEDINA, either by instructing the routing protocol to add this information in routing messages or by multicast messages. Through this mechanism, MEDINA nodes also share pertinent information, such as their capabilities and constraints, throughout their operation, to adapt to variable traffic conditions. The *virtual links* in the MEDINA overlay correspond to possible paths between MEDINA nodes. The path that a packet follows in the overlay is the basis to determine the set of MEDINA nodes that can possibly process it. Moreover, MEDINA nodes constantly monitor the network topology to promptly react to routing changes.

5.2.2 Hash-based coordinated packet selection

During their forwarding operation, MEDINA nodes compute an N bit hash $H(k)$ on a subset k of each packet. We call this subset *hash key*. The packet is captured only if the computed hash value falls within a *specific range* determined through the traffic assignment algorithm. The hash range is different depending on the route the packet is traveling through within the MEDINA domain: generally speaking, the larger the number of MEDINA nodes a flow of packets travels through, the smaller the number of packets each node needs to capture. Moreover, the hash range in each node might be affected by policies and available resources (in the node itself and in other nodes on the path of the packets). Specifically, a MEDINA node determines its hash range as follows:

1. All possible *Ingress-Egress (IE) pairs*[3] in the MEDINA domain are determined starting from the available topological information. If B is the number of edge nodes in a directed graph, the number of IE-pairs is $B(B-1)$.

2. For each IE-pair, the node computes the forwarding path from ingress node to egress node. It is worth noting that, because of the operating principle of

[2]Details of the operations of MEDINA nodes and an analysis of the trade-offs among different options is beyond the scope of this chapter and is left as future work.

[3]While [81] and [76] denote a path as Origin-Destination pair, we prefer the name Ingress-Egress pair to highlight the difference between the original source (destination) of the packet and the ingress (egress) node in the MEDINA overlay.

the shortest path algorithm that is deployed by routing protocols, a packet that is forwarded to its destination through an IE-pair follows the same route as packets coming from the ingress node and addressed to the egress node.

Using the Dijkstra's algorithm the single-source shortest path can be computed with worst-case performance $O(E + V \log V)$ [83] (E and V are the number of links and nodes, respectively). Therefore, the computation for all the IE-pairs can be done in $O\big(B(E + V \log V)\big)$.

3. If the node is *associated* to an IE-pair (i.e., it is included in the IE-pair forwarding path), it computes the range for packets on that path by executing a *traffic assignment function* that, in addition to the number of nodes on the path, takes into account a set of constraints, such as policies and resource availability. Given that the number of IE-pairs containing the node are at most $B(B-1)$ and the number of nodes in a path are at most V, this operation has complexity $O(B^2V)$.

The algorithm for the computation of the hash ranges is summarized in Figure 5.2. The traffic assignment function divides the hash space among all the MEDINA nodes in a path. One possible solution is to split the hash space in different ranges with size proportional to the hardware resources of each node. However, it is possible to devise more complex functions that consider both the hardware capabilities and the expected traffic forwarded by each node. In Section 5.3 we show that even a simple heuristic that considers these two parameters results in a fair load distribution. The outcome of the assignment is a *manifest*: a table that assigns to each IE-pair associated to the node a hash range used to identify the packets forwarded on that path that the node is responsible for capturing. Coherence among the manifests in all MEDINA nodes associated to an IE-pair (namely, associations that avoid redundant capture as well as missing some packets) is ensured by the fact that all nodes on an IE-pair run the assignment function with identical input parameters (other than the specific position of the node in the path). The relatively high complexity of the manifest computation $O\big(BE + BV \log V + B^2V\big)$ (which is $O\big(V^3\big)$ in the worst case) is acceptable given that this operation is performed fully only once; subsequent topology updates require only a partial re-computation that involves only affected paths. If a certain degree of redundancy n must be supported in the capture of packets transiting through an IE-pair, the assignment function ensures that any hash $H(k)$ falls within the hash range of exactly n nodes associated to the IE-pair. MEDINA

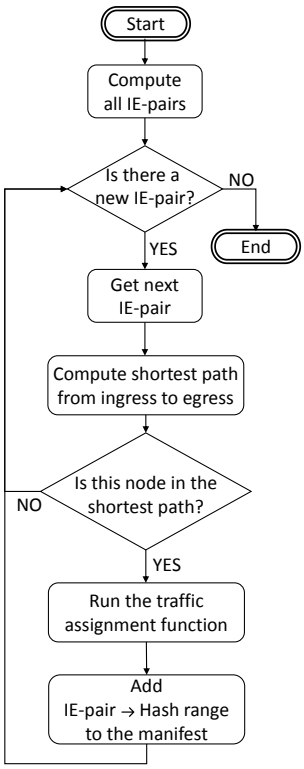


Fig. 5.2 Offline manifest computation

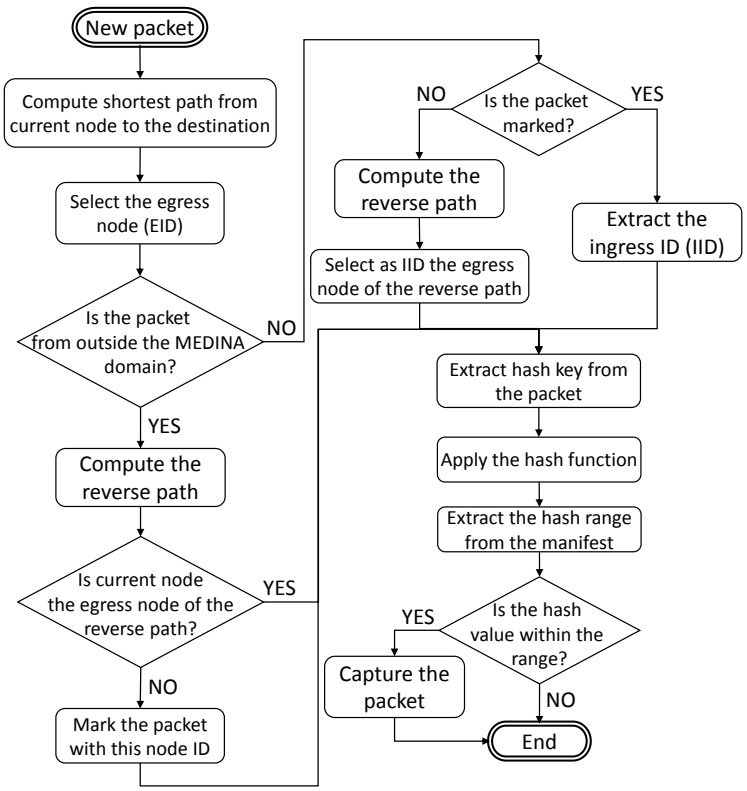


Fig. 5.3 Inline packet processing

offers a valuable contribution when n is much smaller than the number of nodes associated to an IE-pair, which is the case in common deployment scenarios.

Because the hash computed must be the same across all the nodes associated to an IE-pair, the portion of the packet k used as hash key must not change along the path. Thus, in order to achieve a balanced distribution of packets across different hash ranges (i.e., across network nodes), the hash key shall include high entropy, path-invariant fields of the L3/L4 headers and all the bytes of the packet payload, while fields that might be modified on a hop-by-hop basis (e.g., IP TTL, MPLS label) and fields that frequently have the same value in different packets (e.g., IP version) shall be excluded.

Fragmentation by routers internal to the MEDINA domain is not supported since the hash function applied on the whole packet would return different values than on the fragments, which would result into packets captured by multiple nodes or not captured at all. This limitation is compatible with IPv6 networks (where fragmentation is performed by the source) and is practically irrelevant in IPv4

networks (where fragmentation most commonly happens at the network edge [84]). Morever, fragmentation can be avoided through a careful configuration of the MTU in the nodes of the MEDINA network.

5.2.3 Traffic assignment granularity

While the previous discussion has focused on the assignment of individual packets to MEDINA nodes, the selection process can be equally applied to flows, packets or chunks, depending on security and privacy requirements. For example, if the assignment policy specifies all the data carried in one flow to be stored on the same node, flow specific fields of the IP header (i.e., the 5-tuple) can be used as key for the hash function. Some assignment policies might require pre-processing of data extracted from the packet before using it as a hash key. For example, if the data in both directions of a flow must be processed by a single node (e.g., to perform session level analysis), it is necessary to combine source and destination addresses with a commutative function before applying the hash function. In this regard, the XOR function has been demonstrated to have useful properties in terms of ensuring the uniformity of the resulting hash on the hash range [85] and could be applied as follows:

$$(\text{src IP} \oplus \text{dst IP}) || (\text{src port} \oplus \text{dst port}) || \text{L4 protocol}$$

Finally, to have packets acquired in chunks by different nodes, a packet (path-invariant fields and payload) can be split into multiple chunks that are considered separately by the traffic assignment function. In this case, since some chunks do not contain header fields, the entire chunk must be used as hash key.

5.2.4 Path discovery

A MEDINA node must identify the IE-pair each packet is being forwarded through to lookup in the manifest the corresponding hash range and compare it with the packet hash. Since the IP protocol forwards packets according to their destination, given the topology and the destination IP address the node can devise the path the packet will take from itself to the edge of the domain, thus identifying the egress node.

On the other hand, the ingress node a packet has entered the MEDINA domain through cannot be easily inferred. Some approaches are discussed in [86], but they

Algorithm 1 Packet processing algorithm

Require: Node_ID and Manifest

```
1:  p ← new packet
2:  path ← COMPUTE_PATH(Node_ID → p[dst])
3:  EID ← EGRESS_NODE(path)
4:  if p is from outside the MEDINA domain then
5:      r_path ← COMPUTE_PATH(EID → p[src])
6:      if EGRESS_NODE(r_path) ≠ Node_ID then
7:          p[IID] ← Node_ID              ▷ Mark the packet with the Ingress ID
8:      end if
9:  end if

10: if p contains IID then
11:     IID ← p[IID]
12: else
13:     r_path ← COMPUTE_PATH(EID → p[src])
14:     IID ← EGRESS_NODE(r_path)
15: end if

16: key ← EXTRACT_KEY(p)
17: if HASH(key) ∈ Manifest[IID][EID] then
18:     CAPTURE(p)
19: end if
```

rely on information that is usually not available to all network nodes (e.g., ACLs, address block assignments, etc.). As presented in Algorithm 1 (and graphically specified in Figure 5.3), the ingress node of a packet is inferred by considering the egress node of the reverse path (i.e., the path from egress node to source).

- If the ingress node does not correspond to the egress node of the reverse path, the ingress node must provide its identity to internal nodes by marking the packet with a unique *Ingress ID (IID)*.
- If a packet is received without an Ingress ID, internal nodes compute the reverse path and the last hop in the MEDINA domain is assumed to be the ingress node of the packet.

For every packet received from a router outside the MEDINA domain, an ingress node:

1. computes the egress node based on its routing information and the destination IP address,

2. computes the path of a packet in the reverse direction (i.e., from the egress node to the source IP address), and

3. marks the packet with its own Ingress ID if it is not the last hop on such path.

The computation of the shortest path performed by an ingress node in steps (1) and (2) above, as well as an internal node to determine the egress router and possibly the ingress one, has a worst-case complexity $O(E + V \log V)$. Nodes can maintain a table with the IE pairs for the most recent source-destination pairs. Consequently, the shortest path algorithm is executed only for the first packet of a flow. We reasonably expect that the manifest lookup is performed in hardware using TCAMs, thus with time complexity $O(1)$. Also the most recent entries of the IE pair table can be stored in the TCAM to handle the active flows. Moreover, additional optimizations can be introduced in the implementation to reduce the complexity of the algorithm computation per flow and the delay incurred by the first packet of the flow when Algorithm 1 is executed. For example, the egress node for each IP subnet can be pre-computed as soon as the subnet is discovered by routing protocols, i.e., before packets going to or coming from the subnet are received. Such information can be used by internal nodes to devise IE pairs with the complexity of a table lookup, rather than the execution of the shortest path algorithm. This comes to the additional cost of a ternary table containing destination subnets and the corresponding egress routers addresses. If the MEDINA implementation is integrated with the routing protocol implementation, the egress node for each destination can be computed with no added cost when the shortest path algorithm is applied to compute the shortest path to the destination. The egress node can thus be stored in the routing table at the additional cost of one IP address (4 bytes in IPv4) per entry.

Packet marking can leverage unused fields in the IP header (e.g., IP identification field, DS field), network specific fields (e.g., MPLS labels) or tunnelling techniques (e.g., GRE). Given how routing is commonly configured and operated in today's networks, it is reasonable that packet marking is rarely needed. It is worth mentioning that MEDINA can also be deployed in layer 2 networks, such as IXP networks [87], where a path can be identified using source and destination MAC addresses, without requiring packet marking.

5.2.5 Data storage

Since a high traffic rate could result in large numbers of packets captured, an index is built by MEDINA nodes in order to provide fast access to the data. Both the index and the stored traces can be accessed remotely through a set of ad-hoc push/pull APIs. Aggregation of data stored by multiple MEDINA nodes allows for the reconstruction of a detailed and comprehensive image of the overall traffic that can be then further processed and presented through a User Interface (UI). Additionally, every MEDINA node implements a specific eviction policy to manage the available storage space. This policy is used to establish which packets to erase to provide space for the ones being captured. For example, an eviction policy could be to keep a trace for at least the amount of time needed by the system manager to detect an event of interest. In this example, the eviction policy does not guarantee that there is always space available for produced data, hence the available storage space is a parameter that must be considered by the traffic assignment function to assign a smaller portion of the hash space to stressed, resource-limited nodes. Alternatively, a FIFO policy could be used (older data are deleted first), so that new data can always be stored.

5.2.6 Resource allocation

To ensure fair load distribution among the nodes, the traffic assignment function must take into account long-term and short-term traffic dynamics. Different paths transport traffic at different rates and with variable characteristics that affect the capturing effort required by each node. These characteristics vary according to long-term trends (e.g., diurnal traffic is usually higher than nocturnal) and short-term trends (e.g., a one minute traffic burst). Therefore, the parameters used by the traffic assignment function must be adjusted to the variable traffic load forwarded on each path in a certain time period.

Global knowledge of each path's traffic load allows for better load distribution among nodes because it allows nodes to relieve each other from the burden of capturing heavy flows, even if those nodes are not capturing the same flows. For example, if node A is capturing flow 1 and flow 2 and node B is capturing flow 2 and flow 3 (flow 2 is partitioned among node A and B), node B's knowledge of node A's flow 1 allows for node B to take on an additional share (increase its hash range) of flow 2 in case flow 1's processing becomes resource intensive. As a result, node

A has to process less traffic belonging to flow 2 and can dedicate more resources to flow 1. This implies that every node must consider not only the traffic present on its own paths, but also the amount of traffic on the other paths indirectly.

5.2.7 Online fine-tuning

The traffic assignment algorithm guarantees that the load is fairly shared among the nodes assuming that the the allocated resources match the live traffic within a certain tolerance. However, to cope with less predictable short-term traffic variations, a MEDINA node constantly monitors the resources used by each path to notice any significant deviation from the allocated amount (both in the case of excessive or scarce traffic).

When the current traffic and the allocated resources differ by a value above a given tolerance threshold, the allocation plan is re-evaluated to adapt to the current state. As a result, this triggers an update to the local constraints which are then sent to all the nodes involved in the relative path, triggering an update of the manifests throughout the network. The tolerance must be selected as a trade-off between the level of fairness in the load distribution and the frequence of the updates (influencing the amount of additional synchronization traffic). Additionally, it should also consider the time needed to disseminate the updates, which can vary with network congestion.

Moreover, MEDINA nodes constantly monitor the network topology (piggybacking on routing protocols or, when this is not possible, through ad-hoc keepalive messages) to promptly react to routing changes. Failures of MEDINA nodes are immediately detected, so that the remaining nodes can re-evaluate the allocation plan to divide among them the hash ranges of a failed node.

5.3 Evaluation

In order to evaluate the benefits provided by MEDINA in a realistic scenario, we simulate its deployment on the *Internet2* network, considering its publicly available PoP-level topology and static link weights. In our experiment in each PoP there is a MEDINA node and the amount of storage in the node is proportional to the population of the city that it serves with a ratio of 1 GB for each 100 people (from 2 TB for Salt

Lake City to 84 TB for New York). We expect that in a real deployment populous cities would be served by multiple nodes, but we preferred to keep the simulation simpler while still having the algorithms operate in an equivalent situation.

Reproducing the evaluation performed in [81], we apply a gravity model to a baseline traffic volume T_b of 8 million IP flows per 5-minute interval to obtain the traffic volume $TV_{i,e}$ for each IE-pair (B is the set of edge nodes and P_j is the population served by the edge node j):

$$TV_{i,e} = T_b * \frac{P_i * P_e}{\sum_{j,k \in B} P_j * P_k}$$

Specifically, we assume that the total traffic between 2 PoPs is proportional to the product of their population sizes. We assume that the flow size (number of bytes per minute transferred) is Pareto-distributed with shape 0.606 and scale 92 [88] and that a single flow cannot transfer over 750 MB per minute (corresponding to a 100 Mbps rate).

The traffic assignment function implements a simple heuristic to distribute the load among the nodes considering for each one of them: (*i*) the amount of available storage, (*ii*) the number of IE-pairs the node is associated to, (*iii*) the expected traffic through each associated IE-pair (based on the forwarded traffic). Specifically, the storage available to a node is divided among the associated IE-pairs proportionally to the expected traffic in each IE-pair. The result of this allocation is a set of constraints that are shared with the other nodes involved in the same IE-pair. Finally, the hash space for the IE-pair is divided among the nodes in the path proportionally to the constraints advertised by them. Moreover, each MEDINA node constantly monitors the used storage space and, whenever it concludes that the available space is reducing excessively fast, it recomputes the contraints (with new traffic estimates) and sends a message to the other MEDINA nodes associated to the same IE-pairs to recompute a more fair assignment. As a result, if traffic through the various IE-pairs matches the expectation (within a predefined threshold), the algorithm converges to a steady state that does not require further communication among the nodes. Exceptional changes in the traffic characteristics trigger the computation of new constraints to converge to a different distribution plan.

Figure 5.4 and Figure 5.5 present the results of this simulation. Figure 5.4 shows that, even with our simple heuristic, most nodes' used storage increases at

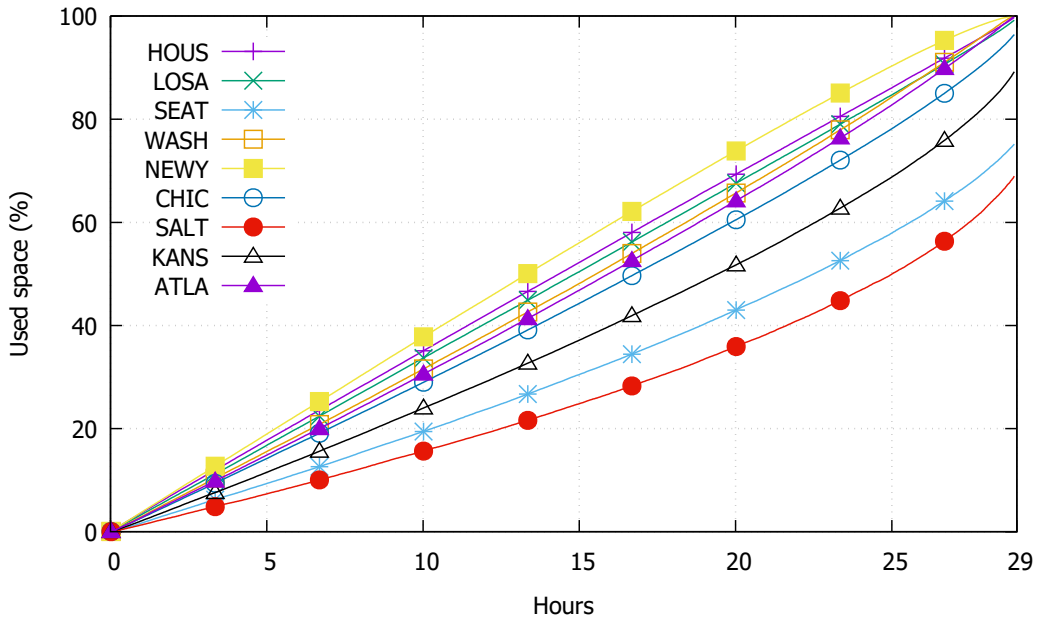


Fig. 5.4 Per node used storage.

a similar pace over time. However, as shown in Figure 5.5, some nodes forward less traffic than others, hence nodes cannot all capture the same fraction of their traffic. Specifically, as the storage space becomes a critical resource, these nodes take a larger share of the traffic they forward compared to nodes traversed by a larger number of flows, thus lessening the capture burden on the latter. Even with this simple heuristic, it takes around 29 hours to reach a point where all the nodes in a path are completely full and successive flows cannot be captured. As a result, in this scenario the captured traffic can be available to the NOC for offline processing (e.g., network forensics) for more than a day. Presumably, with that amount of time, the raw data of interest can be retrieved by the NOC during periods of limited network activity.

Figure 5.5 shows the amount of traffic that each node forwards and the fraction they capture. On average a node captures only around 25% of the traffic it forwards, thus MEDINA allows to reduce by a factor of 4 the amount of resources needed for processing and storing the traffic in each node compared to the case of capture happening independently in selected nodes. Consequently, MEDINA also allows reducing the amount of additional traffic in the network if all the captured packets are to be sent to a NOC. Moreover, the NOC does not have to identify and remove duplicated packets. It is worth noting that in a real scenario the gain is even greater

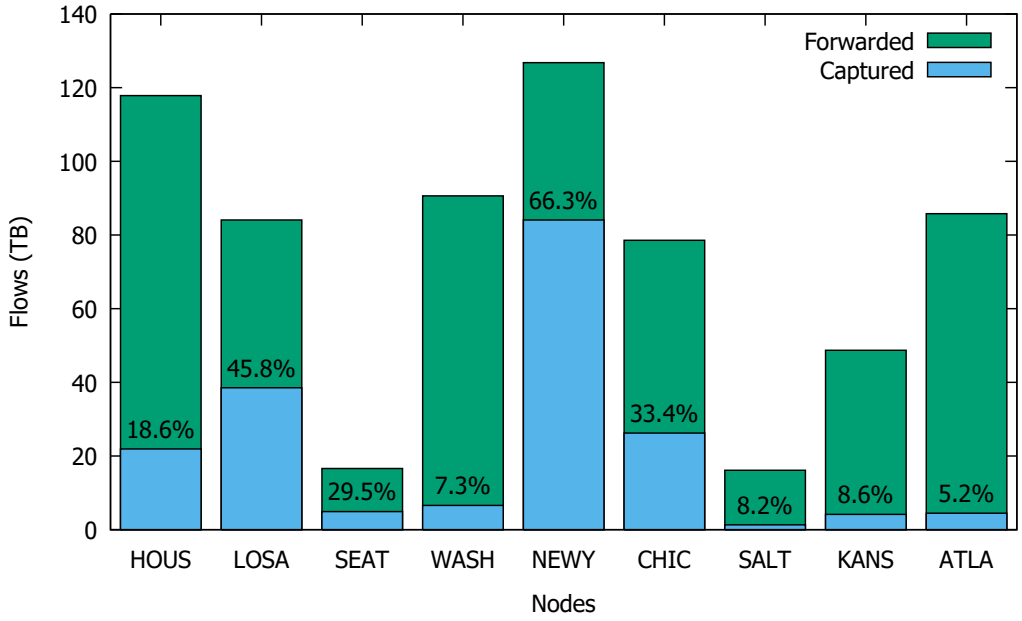


Fig. 5.5 Per node captured and forwarded traffic.

than in our experiment where a PoP-level topology is considered: core routers could also support MEDINA and share the traffic capture burden.

A more complex heuristic could provide a more equal load distribution among the nodes, but it also would come at the cost of a larger amount of information exchanged among the nodes and a longer computation time to obtain the assignment.

5.4 Related Work

A number of techniques have been proposed to distribute the traffic processing load among network devices in a coordinated fashion, especially for monitoring tasks. The work presented in [89] proposes to distribute monitors in the network that determine the monitoring target by periodically sharing information on the monitored flows. This information is shared as an aggregate by means of counting bloom filters. This solution aims at reducing duplicate monitoring, however, since frequently the first monitor in the path is responsible for the flow, the load is not fairly distributed among all the monitors. Moreover, since each monitor must be constantly aware of the flows others are responsible for, bloom filters must be exchanged frequently,

as well as stored and processed by each monitor, which limits the scalability of the solution.

Sonata [90] proposes an SDN approach to network monitoring, where a controller defines a set of flow rules used to sample the traffic processed by various network devices. The samples are sent to a centralized stream processor, which analyzes the data and refines the sampling policy. This approach is applicable for monitoring persistent events, such as Denial Of Service attacks. In fact, the iterative refinement allows to zoom-in on successive sets of packets, assuming that the features of interest continue to be present in the traffic. On the contrary, MEDINA allows to zoom-in also on past, terminated, events, given that the processed traffic is stored and indexed by network devices themselves.

Many of the solutions that provide coordinated distribution and load balancing are based on a centralized controller, possibly running in the NOC, responsible for setting the target traffic for each node. In cSamp [81] a centralized system assigns sampling responsibilities to routers to optimize network-wide monitoring objectives. As in MEDINA, each router has a table with a hash range for each path on which it lies. The router computes on each packet the hash of the 5-tuple and processes the flow only if the hash falls in the range associated to the path followed by the packet. However, cSamp requires to mark all the packets upon entering the network with an identifier of the path they take in the network, while MEDINA does not require any additional information to be included in most packets (i.e., those for which the ingress node corresponds to the egress node of the reverse path, which is the case for the vast majority of packets since routing is normally symmetric). An updated version of cSamp is presented in cSamp-T [91], where packet marking is limited at the cost of possible duplicate processing and sub-optimal results in terms of load distribution and coverage of packets. In cSamp-T, instead of using per-path hash ranges, routers deploy hash ranges that depend on a 3-tuple consisting of previous hop, current router, and next hop. Since the 3-tuple can be inferred using only local information, packet marking is not required. However, it is not possible to control the number of hash ranges a packet traveling through the network will fall into. As a results, a packet might be processed more than once or not at all, which makes cSamp-T applicable only in applications involving flow sampling. It is worth noting that the technique used to split responsibilities across multiple network nodes proposed in cSamp has been deployed also for WAN optimization [92], which proves

that the approach (whose underlying principles are similar to the ones of MEDINA) can be successfully applied to different traffic processing services.

Leisure [76] deploys a centralized architecture that uses global network-wide information to allocate disjoint sets of flows to be measured. The framework distributes traffic measurement tasks evenly across coordinated routers, leveraging a heuristic aimed at minimizing the variation of monitoring load among nodes. MEDINA nodes instead share the load fairly according to node capabilities, therefore more powerful nodes can receive a larger share, relieving other nodes from the load on a set of paths, providing the opportunity to use their resources on other paths. The authors of [77] apply a coordinated approach to distribute remote packet capture. In the proposed solution the controller increases the number of sensors in the network only when the load grows and the deployed sensors are close to their utilization limit, thus does not favor a fair load distribution.

In wide-area deployments centralized approaches suffer from the additional latency required to convey topology and resource utilization information to the centralized controller and to communicate the new allocations produced by the controller back to the various nodes. This, as discussed in Section 5.1, prevents prompt reaction. A decentralized coordination for flow monitoring is introduced with Decon [93], where a peer-to-peer overlay of multiple controllers, called rendezvous points (RPs), is responsible for monitoring decisions on new flows. Specifically, Decon's RPs decide which probes in the network should monitor which set of flows going through them. The objective is to increase coverage, in terms of number of flows monitored during a given time period, by spreading the load across the available resources. Even though the control is decentralized, the controller and the monitor are separate entities, thus each monitor must actively report to a RP the detection of new flows to request an assignment decision, with a considerable amount of control traffic in the network on a large scale. A similar solution is presented in Decor [94] where egress nodes decide the optimized resource arrangement. Kamiyama et al. [95] propose an autonomous load-balancing method for flow monitoring where monitors exchange information on their load only with adjacent monitors. Hence, each node independently adapts the monitoring target based on data received from its neighbors only, which favors a local optimum. Instead, in MEDINA constraints are exchanged with all the nodes in the overlay; however, this is done only when the load exceeds the estimate in order to contain the amount of service traffic.

5.5 Conclusions and future work

This chapter presented MEDINA, a highly distributed and decentralized traffic capture and processing platform. Although the principles underlying MEDINA could be applied to generic processing and storage of data units forwarded through a network, this work focuses on traffic monitoring, with the goal of fairly distributing the resulting load among nodes through decentralized coordination. The load distribution is autonomously adapted to changing traffic conditions, leveraging data shared by all the nodes in a path. Different security and privacy requirements are met by selecting a custom granularity in the traffic assignment to individual nodes that can operate on a per-flow, per-packet and even per-packet chunk basis. Moreover, nodes process and store traces and metadata with a configurable amount of redundancy for increased reliability. A distributed index, queried through ad-hoc APIs, provides fast access to the stored data for offline processing and analytics. With the experimental evaluation of a naive implementation of MEDINA we show that in realistic scenario it is possible to reduce by a factor of 4 the amount of processing required in each node, compared to independent packet capture.

As future work we plan to analyze the impact of extending the approach to generic processing (beyond packet capture) in a network of generic entities (rather than routers) and to study the tradeoff between the achievable load balance, the amount of additional information exchanged by the nodes and the computation time needed to define the distribution plan. Future work will also be aimed at proving the theoretical correctness of the distribution algorithm and finding an optimal traffic assignment function that minimizes the variation of load among MEDINA nodes.

Chapter 6

In-network computation with programmable data plane

6.1 Introduction

With flexible networking hardware [97] and expressive data plane programming languages [98, 99] the functionality of networks can now be enriched without hardware changes while retaining the capability of processing packets at very high rates, even above Terabits per second. Emerging programmable network devices are paving the way for new services to better support data center applications [100, 101] and improve network monitoring [102–106].

Programmable networks create the opportunity for in-network computation, i.e., offloading a set of compute operations from end hosts into network devices such as switches and smart NICs. In-network computation can offer substantial performance benefits, as it is for example the case with consensus protocols [107, 100] and in-network caches [108, 109]. Although traditional networks are not capable of computation, the idea of using the network not just to move data, but also to perform computation on transmitted data is reminiscent of *Active Networks* [110], which proposed to replace packets with small programs called "capsules" that are executed

The content of this chapter has been published in [96].

at each traversed switch. However, for the past two decades the hardware capabilities were lacking. This appears to be changing.

The recently proposed RMT architecture [97] and its upcoming incarnation in the Barefoot Networks' Tofino [111] switch chip has a flexible parser and a customizable match-action engine. To process packets at high speed, this architecture has a multi-stage pipeline where packets flow at line rate. Each stage has a fixed amount of time to process every packet, allowing for lookups in memory (SRAM and TCAM), manipulating packet metadata and stateful registers, and performing boolean and arithmetic operations using ALUs. Other vendors are also introducing new classes of programmable chips with similar capabilities [112]. We believe that with this new generation of flexible data plane hardware it is worth revisiting a fundamental question: *as networks become capable of computation, what kinds of computation should networks perform?*

In this chapter, we will consider this question in the scope of data center applications because it is likely that data centers will be early adopters of programmable networks and many of these applications have stringent performance requirements. On the one hand, in-network computations can be broadly useful in several performance-oriented contexts to reduce latency and/or increase throughput of certain operations. Furthermore, it can help reducing network traffic, so as to alleviate congestion, which is a major cause of application performance degradation. In particular, a computation that happens on-path and at line rate is appealing since it bears no cost to the application, which can spare CPU cycles for other tasks instead. On the other hand, despite recent technological advancements, network devices have limited compute power and little storage to support general computation. Moreover, systems designers are prescribed by the end-to-end principle [113] to avoid implementing application-specific logic in the network and are generally wary about raising the overall system complexity. Additionally, in-network computation must not affect the application correctness.

We posit that in-network computation must be used judiciously. Towards this goal, we seek to identify *what type of computation can be done in-network* such that: (*i*) network traffic is significantly reduced, (*ii*) only a minimal change at the application level is required and (*iii*) the correctness of the overall computation is not affected.

We find that a plausible class of applications that satisfy the above desiderata are those applications that follow a partition/aggregate workload pattern. These applications cover a wide spectrum of data-intensive frameworks including big data analytics as in MapReduce [114] and machine learning [115–117], graph processing [118, 119] and stream processing [120]. Generally these frameworks scale applications by distributing data and computation across many worker servers. Each worker performs some computation on a data partition, which is followed by a communication phase to update shared state or finalize the computation. This process can be performed iteratively until a stopping condition is met.

These applications are sensitive to network performance and the communication cost can be one of the dominant scalability bottlenecks as large volume of data need to be moved routinely in many to many patterns. Already several distributed frameworks like MapReduce [114], Pregel [118] and DryadLINQ [121] allow for *user-defined aggregation functions*. These functions enable application developers to reduce the network load (e.g., by summing all individual messages to a common graph vertex) and consequently, the job execution time. However, the aggregation functions are only applied at the worker-level, missing the opportunity of achieving better traffic reduction ratios when applied at the network level.

These aggregation functions have several characteristics that make them appealing and suitable to be partially executed in-network. First, they usually *reduce the amount of data* (e.g., sum the inputs, or find the minimum). Thus, it is beneficial to apply these functions as early as possible to decrease the amount of network traffic and lessen congestion. Second, they are usually characterized by *simple arithmetic/logic operations*, which make them amenable to parallelization and execution on programmable switches. Third, in many algorithms [118, 119], they are *commutative and associative functions*, which implies that they can be applied separately on different portions of the input data, disregarding the order, without affecting the correctness of the final result. Fourth, they are *often readily available*, meaning that they could be transparently supported without requiring the developer to write new application logic. As we will show, implementing certain aggregation functions inside the network is possible and beneficial since programmable switches can aggregate intermediate data, thus reducing the traffic as well as the processing load at the destination. However, the limited resources, restricted compute power and stringent constraints on packet processing time create several challenges and call for a judicious system design.

To contribute a concrete point in the design space, we propose DAIET, a system for data aggregation in-network. While our design is still incomplete and likely to change, this represents an example of a system that can be built using the P4 programming language [98] to offload computation to the data plane. Our experimental results with an initial prototype supporting a MapReduce application show that this approach provides a large data reduction (86.9%-89.3%) and a similar decrease in worker's computation time.

A number of recent research efforts [122–124] have proposed in-network aggregation techniques for a variety of applications. However, these systems either required to change the network architecture [122, 124] or build a switch chip with a fixed set of aggregation functions [123]. We demonstrate that similar benefits can be reaped using flexible and programmable data planes. That said, we envision that practical deployments for our proposal might be better suited within clusters and racks specialized for certain workloads such as deep learning or data analytics where the benefits of in-network aggregation are substantial without requiring data center-wide adoption.

We note that aggregation functions, though they are a generic primitive applicable to a number of applications, are not the sole type of in-network computation possible and we hope that our work will trigger a broader discussion around the driving question behind our work: what should networks compute?

6.2 Background

The idea of using the network not only as a passive mover of bits, but also as a more general computation engine, where information injected into the network may be modified, stored, or redirected, has been the subject of extensive research within the field of *Active Networking* [110]. Active networks proposed to let the user choose the functions that the network would perform on its traffic by inserting directly in the packets the specific function ID and the required parameters. We advocate that this approach can be revisited thanks to the introduction of flexible dataplane hardware.

Many data center applications scale by combining phases where work and input data are partitioned across many independent worker threads executed in parallel by different servers with phases where partial or intermediated results are aggregated

by one or many servers and are subjected to further processing, in order to compute the final set of results. One example of such applications is MapReduce [114], a data-parallel programming model designed for scalability and fault-tolerance that scales to large data volumes on thousands of machines. The same pattern appears in a variety of applications like graph processing [118], machine learning [116, 117], stream processing [125], and others.

The distribution of intermediate results requires one or several many-to-one communications among servers, which stress the underlying network, given the sheer amount of data and possibly oversubscription within the network. Moreover, the large number of uncorrelated flows in many-to-one communications may overrun the buffers of top-of-rack switches (TCP *incast* [126, 127]), causing TCP throughput to drop, which in turn increases job completion times.

In many cases, the aggregation phase applies application-specific aggregation functions that reduce the amount of data (e.g., sum the inputs, or find the minimum). Thus, it is beneficial to apply these functions as early as possible to decrease the amount of network traffic. Moreover, in many algorithms [119], the functions are *commutative* and *associative*, which implies that they can be applied separately on different portions of the input data, disregarding the order, without affecting the correctness of the final result. These properties are crucial to perform in-network aggregation.

6.2.1 P4 Programming Language

P4 [98] is a data plane programming language to define customizable packet processing in network devices. It has been designed around three goals: (*i*) protocol independence: new protocols can be supported without hardware modifications; (*ii*) target independence: starting from a generic P4 program, a compiler generates the final code specific for a particular target platform (e.g., reconfigurable hardware switches [97, 128], smart NICs [129] or software switches); and (*iii*) reconfigurability: packet parsing and processing can be redefined in the field.

As in a classical SDN scenario, the control plane can configure the P4 data plane by pushing *MATCH-ACTION* flow rules in a set of tables. These flow rules can match custom protocols and execute custom compound actions. Actions can modify the packet, update local state variables and then drop the packet or forward it to

the next hop. P4 actions can also clone packets (e.g., to support multicasting) and resubmit it to the ingress port for a new round of processing.

A P4 program is made of five constructs: (*i*) packet *headers* defining the order and the size of data carried by the packet; (*ii*) *parsers*: stating how to transform packets to a parsed representation by specifying the sequence of headers that can be present in a packet; (*iii*) *tables*: describing the header fields that the rules in these tables can use to select packets. They also describe how these fields are matched (e.g., exact match, Longest-Prefix Match, etc.) and the list of possible actions performed on matching packets; (*iv*) *actions*: defining a sequence of primitive operations; e.g., add or remove headers, modify headers fields, update the local state, clone, drop or packet forwarding; and finally (*v*) *control blocks* specifying the sequence of tables that must be applied to each packet. Unlike typical SDN approaches (e.g., *OpenFlow* [130]), flow rules cannot state to submit packets to a different table (i.e., `goto-table` action), only control blocks can select the tables to apply.

P4 programs can keep per-packet state by defining a list of *metadata*. These metadata are discarded as soon as the packet leaves the switch, while persistent state is kept using *registers*, which are byte arrays whose size must be defined at compile time.

6.3 Judicious Network Computing

We focus on in-network computation enabled by the recent developments in reconfigurable, protocol-independent switch ASICs such as RMT [97]. Their network machine architecture is based on a multi-stage pipeline of packet processing logic [131]. Computing on these devices corresponds to executing streaming algorithms that have stringent constraints on the number and type of operations that can be performed. This is due to the following limitations:

Limited memory size. Packet processing at high speed requires a very fast memory, such as TCAMs or SRAM, which is expensive and usually available in small capacities. As an example, the upcoming Barefoot Tofino [111] switch chip is expected to process a remarkable 6.5 Tb/s while still providing the flexibility of data plane programmability. To match this processing speed, packets can be processed by a

limited number of lookup tables and the expected available SRAM is in the range of few tens of MBs.

Limited set of actions. Programmable devices support a small set of actions, usually simple arithmetic, data manipulation, and hashing operations. Some switches can also provide limited support for floating point operations [123].

Few operations per packet. To guarantee execution at line rate, programs have only tens of nanoseconds to process a single packet. As a result, they cannot use constructs that do not have an upper bound on the number of performed operations (e.g., loops). Some devices allow to recirculate a packet in the ingress queue for further processing, thus allowing to implement loops. But this comes at the cost of additional processing latency and lowers the forwarding capacity.

Furthermore, offloading functionality to the network faces several challenges. First, the underlying target imposes restrictions as discussed above and resources are limited. As such, in-network computation must live within these confines. Second, applications correctness is paramount. Data center networks have multiple paths and failures of links and devices are not uncommon. As such, in-network computation should provide benefits even if traffic follows different paths or an application experiences failures. Alternately, an application should be no worse than without in-network computation even when this is executing. Third, offloading functionality raises complexity not only because certain packet processing logic is executed in the network but also due to the required integration with applications or libraries. As such, in-network computation should focus on primitives that are broadly applicable to a class of applications and workloads, and identify reusable, high-level abstractions that promote easy adoption.

Since we are just at the onset of programmable data plane hardware, it is hard to gauge how far in-network computation can go and what possibilities the next technological enhancements will enable. As an analogy, the spectrum of applications of GPUs has evolved significantly since when they first became programmable, and today's GPGPUs have usages for deep learning and mining cryptocurrencies, that are far beyond the computer graphics domain.

In the rest of this chapter, we follow through our earlier premise and explore in-network aggregation as a concrete example of in-network computation. Consequently, a main challenge to address when delegating data aggregation to the network is to

account for all the constraints above while providing high data reduction ratios without affecting the correctness of the final results.

6.4 Data Aggregation in Data Center Applications

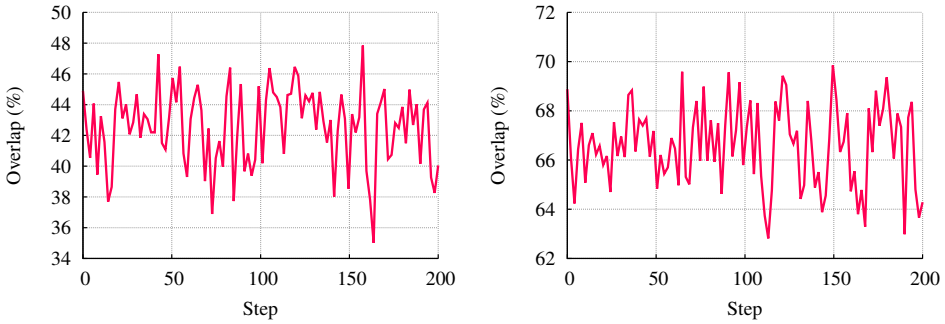
Data aggregation is a common task in several distributed data center applications [114, 117, 118, 125]. It also satisfies the characteristics discussed in Section 6.1. Therefore, it represents a good candidate for tasks that can be delegated to the network. In this section, we study a set of algorithms that can utilize aggregation functions to improve their performance. Specifically, we consider two classes of algorithms: machine learning and graph analytics. The goal of this analysis is to show the potential traffic reduction that can be achieved when aggregating the traffic inside the network.

For machine learning algorithms, we use TensorFlow [117] to run two applications: a Soft-Max Neural Network using mini-batch Stochastic Gradient Descent (SGD) and Adam optimization [132] (Adam). We use a mini-batch of size 3 for the former and 100 for the latter. In these experiments, we use the MNIST[1] database of handwritten digits. The model is trained to correctly identify the digits present in each image. We deployed TensorFlow on six machines: one acts as the parameter server while the other five machines run as many worker processes. Each machine is equipped with 128GB of RAM and two 2.20GHz Intel Xeon E5-2630v4 CPUs.

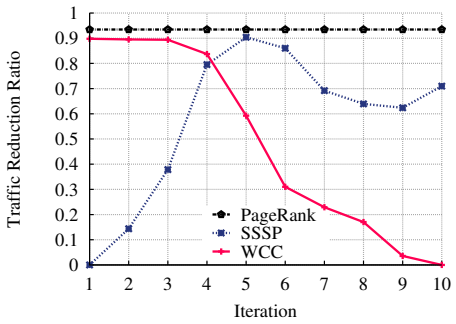
Workers are responsible for compute-intensive tasks while the parameter server stores and maintains a set of shared parameters that comprise the trained model. In this setting, each worker is training the same model on different mini-batches of the data. In each iteration, the worker sends its parameter updates to the server which aggregates the local updates from each worker. Then, the parameters at each worker are updated according to their values at the parameter server.

In TensorFlow, the parameters are tensors, which are represented as large n-dimensional arrays. Parameter updates are deltas that change only a subset of the overall tensor and can be aggregated by a vector addition operation. We evaluate the overlap of the tensor updates, i.e., the portion of tensor elements that are updated by multiple workers at the same time. This overlap is representative of the possible data

[1] http://yann.lecun.com/exdb/mnist/



(a) Stochastic Gradient Descent (b) Adam optimization



(c) Graph Analytics Algorithms

Fig. 6.1 Potential traffic reduction ratio for two machine learning applications and various graph analytics algorithms.

reduction achievable when the updates are aggregated inside the network. A high overlap means that aggregating the local updates of each worker inside the network could reduce the network traffic significantly.

Figures 6.1a and 6.1b show the amount of overlap among workers updating the same portion of tensors in the same iteration for SGD and Adam applications, respectively. Note that the overlap percentage is consistent among different iterations. The average overlap percentage is around 42.5% and 66.5% for SGD and Adam applications, respectively. Also note that the results in this experiment represent a lower bound of the possible overlap as the applications could be tuned to schedule communication to maximize overlap. In both applications, there are other tensors communicated over the network with a higher overlap percentage, or even fully communicated (100% overlap). We also experimented while increasing the number of workers from two to five (without changing the mini-batch size), and observed that the overlap increases.

We further consider graph analytics algorithms. We used the LiveJournal dataset,[2] which consists of 4.8M vertices and 68M edges. To run these algorithms, we deployed GPS [133] – an open-source Pregel clone – on four machines, each with 3.40GHz Intel Core i7-2600 CPU and 16GB of RAM. We consider three algorithms with various characteristics: PageRank, Single Source Shortest Path (SSSP) and Weakly Connected Components (WCC). The three algorithms are associated with a commutative and associative aggregation function.

Figure 6.1c shows the potential traffic reduction ratio for various graph algorithms using the LiveJournal graph. Each graph algorithm exhibits a different traffic volume. In PageRank, each vertex starts by sending its PageRank value to all its neighbours. Then, each vertex in the next iteration receives and sums the various values from its neighbours and calculates a new PageRank value. The traffic reduction ratio is calculated by combining all the messages sent to the same destination into a single message by applying the aggregation function used by the algorithm, i.e., *sum*, inside the network. In each iteration, all vertices are active and send messages to their neighbours; hence, the traffic reduction ratio is almost the same across all iterations. SSSP starts by sending a smaller number of messages from the source vertex. In the following iteration, the number of messages increases exponentially and hence a higher traffic reduction ratio is achieved. On the other hand, WCC starts by sending large number of messages from all vertices which decrease as the algorithms converges. The potential traffic reduction ratio in all the three applications ranges from 48% up to 93%. In summary, applying in-network aggregation functions could significantly reduce the traffic of these applications.

6.5 Solution sketch

As a proof-of-concept, we propose DAIET, a system for in-network aggregation designed to address the challenges presented in Section 6.3 [134]. While it has been designed with P4 and programmable ASICs in mind, it is general enough to be possibly implemented on different programmable data plane platforms. For the sake of presentation, we describe DAIET when applied to MapReduce-based applications. However, the proposed solution is generic enough and works well for various partition/aggregate data center applications.

[2] https://snap.stanford.edu/data/

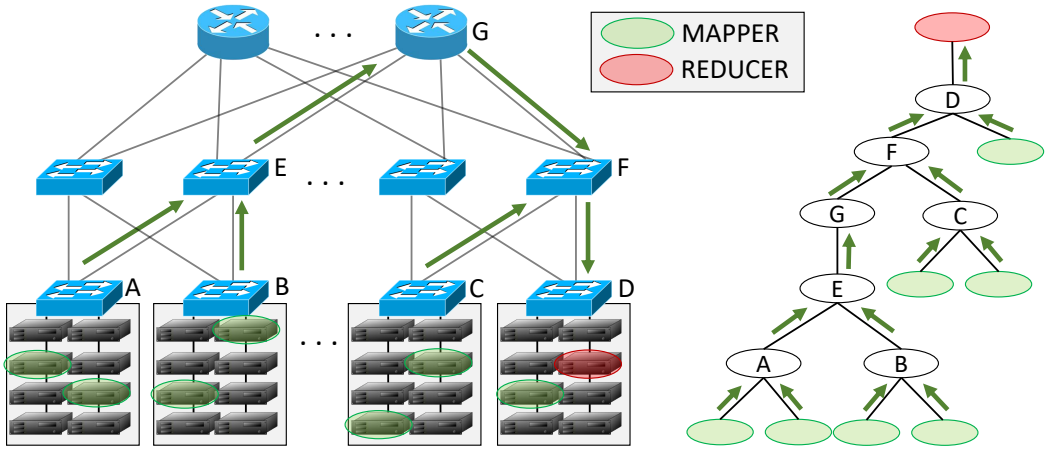


Fig. 6.2 Aggregation Trees: example of physical and logical view for traffic aggregation in a data center network.

To perform in-network aggregation, DAIET requires a close collaboration between the application and the network. Prior to starting a job, the master allocates the *map* and *reduce* jobs to the workers. This allocation information is exchanged with the network controller. Then, the controller defines the aggregation trees. An aggregation tree (Figure 6.2) is a spanning tree covering all the paths from all the mappers to a reducer. There is one tree rooted at each reducer. The network controller then configures the network devices, pushing a set of flow rules, to perform the per-tree aggregation and forward the traffic according to the tree. Specifically, each network device needs to know (*i*) the *tree ID* (i.e., *reducer ID*) that is embedded in packets addressed to one reducer, (*ii*) the associated output port to forward the traffic to the next node in the tree, and (*iii*) the specific aggregation function to perform. Moreover, for each tree a network device is configured with the number of children nodes it receives traffic from, so that the aggregated data are flushed to the next node when all the children have sent their intermediate results.

When the map phase is completed, each map task produces an intermediate data set consisting of key-value pairs, which is partitioned among the reducers. These partitions are sent to the reducer using UDP packets containing a small preamble and a sequence of key-value pairs. In the current prototype, we do not address the issue of packet losses, which we leave as future work. The abstraction of key-value pairs can be mapped on the messages exchanged in various data center applications; e.g., they can represent updates to shared parameters in a machine learning job or exchanged messages among vertices in graph processing.

The preamble specifies the number of pairs present in the packet and the tree ID the packet belongs to. We have carefully defined how the output of the map task is serialized in the local file, so that packets are transmitted without partial pairs. In fact, data cannot be deserialized during packetization, since it would greatly affect the execution time, therefore we use a fixed-size representation for the pairs, so that it is easy to calculate the offsets of pairs in the file and extract a number of complete pairs. This serialization and deserialization modification is not required in those applications that do not store intermediate results on disk. Finally, the end of the transmission is marked by a special END packet.

For each tree, network devices store two arrays, one for the keys and one for the values. These two arrays are managed as a hash table with buckets of only one element. Specifically, a hash function is used to convert a key to an index in the array. The index is used to access the two arrays and store the key and its corresponding value in the relative cells. If a collision is detected (a key generates the same hash of a different, already stored, item), then the new pair is not used for aggregation, and is stored in a different *spillover bucket*. This bucket is a queue of pairs with as many entries as the number of pairs that can fit in one packet. When this bucket is full, the entries are immediately sent to the next node in the tree. If the hash function distributes hash values evenly across the available range, this solution better employs the available memory (a scarce resource in data plane devices) without affecting the correctness of the final result. In fact this solution saves the allocation of multiple collision buckets that have a low probability to be used. The non-aggregated values in the spillover bucket are the first to be sent to the next node, so that they ware more likely to be aggregated if the next node is a network device and has spare memory. Additionally, an *index stack* is kept in the device memory to store the indices of the used cells in the two arrays. This facilitates flushing the results to the next node, avoiding a costly scan of the arrays.

Algorithm 2 summarizes the steps performed by the network device for each received packet. When a new packet containing key-value pairs is received, each pair in the packet is processed to update the local state. First, the hash function is applied to the key to obtain the corresponding index. This index is then used to access the keys array and check if: (*i*) the cell is empty, (*ii*) the same key has been received before, or (*iii*) a different key with the same hash value is already stored in the cell. In the first case, the new key-value pair is stored and the index is saved in the index stack. In the second case, the value is aggregated with the previously stored value

Algorithm 2 Packet Processing Algorithm

```
Require: Network Packet P
 1: header ← PARSEHEADER(P)
 2: if header.type = DATA_PACKET then
 3:     entries ← PARSEPAYLOAD(P, header.num_entries)
 4:     for all pair ∈ entries do
 5:         idx ← HASH(pair.key)
 6:         if keyRegister[idx] is empty then
 7:             keyRegister[idx] ← pair.key
 8:             valueRegister[idx] ← pair.value
 9:             indexStack.PUSH(idx)
10:         else if keyRegister[idx] = pair.key then
11:             UPDATEVALUE(valueRegister[idx],pair.value)
12:         else
13:             STORE(spilloverBucket,pair)
14:             if spilloverBucket is full then
15:                 FLUSHDATA(spilloverBucket)
16:             end if
17:         end if
18:     end for
19: else if header.type = END_PACKET then
20:     remaining_children = remaining_children - 1
21:     if remaining_children = 0 then
22:         FLUSHDATA(keyRegister, valueRegister)
23:     end if
24: end if
```

and the result is stored in the array. In the latter case (i.e., a collision), the pair is stored in the spillover bucket. If this bucket is full, all its pairs are sent to the next node.

When an END packet is received, marking the end of one partition, the number of pending children (initialized by the controller) is decremented. When this value reaches zero, all the aggregated pairs in the two arrays can be sent to the next node towards the destination.

While usually a reducer receives the intermediate results from each mapper sorted according to the key, DAIET cannot preserve the order, thus the reducer receives unordered, aggregated, intermediate results. As a consequence, the intermediate results must be sorted at the reducer rather than at the mapper, which usually reduces

the amount of parallelism. However, as shown in Section 6.6, the reduction in the amount of data to sort makes this overhead negligible.

6.6 Preliminary Evaluation

Our current implementation of DAIET is built using P4 and is available as open source[3]. As in a traditional SDN approach, the controller can configure a P4 data plane by pushing flow rules to a set of tables. These flow rules can match custom protocols and execute custom actions. We found that P4 imposes two main constraints affecting the implementation of DAIET: (*i*) a table can be applied at most once per packet, therefore it is not possible to apply the same table to all the headers in a stack of multiple headers of the same type, and (*ii*) the absence of variable-length data structures.

The first constraint, which is meant to avoid loops during packet processing, forces the programmer to manually perform loop unrolling, at the expenses of code readability and size. The second constraint is relevant in case of variable-length keys (e.g., strings). In fact, in this case the programmer is forced to reserve for each key as many bytes as the largest expected key, increasing the memory footprint, which in turn causes the allocation of arrays with fewer cells, thus increasing the possibility that a pair is not aggregated.

We present preliminary results from our prototype implementation. We focus on quantifying the reduction of traffic received by the root node of each tree (i.e., reducers) and the corresponding decrease in completion time at reducers. We believe this reduction, in a deployment with hardware switches, is expected to be proportional to the reduction in the job completion time, since each reducer will receive and process less data. However, as P4 hardware was not yet available to us, we obtain results using the bmv2 software switch,[4] which is not designed for line-rate packet processing. Thus, we cannot directly measure an improvement in job completion time but our results, which show around 88% median traffic reduction are still indicative of the expected benefits.

[3] https://sands.kaust.edu.sa/daiet/
[4] https://github.com/p4lang/behavioral-model

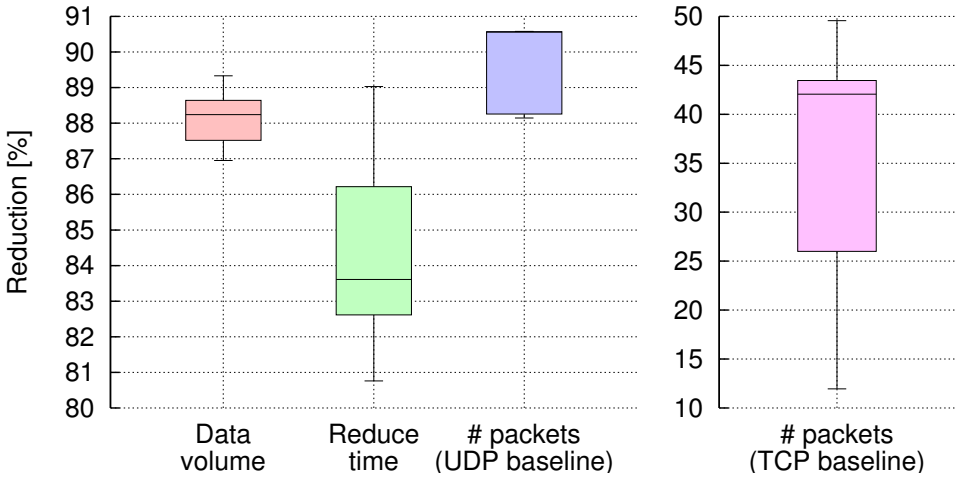


Fig. 6.3 Reduction on the amount of data, running time and number of packets received at reducers.

We run the experiments on a single server with two Intel Xeon E5-2680v2 CPUs with 40 logical cores in total and 768 GB of RAM. We run the bmv2 switch in a container on 4 dedicated cores, while 12 more containers, each with 2 dedicated cores, are used as workers to run 24 mappers (one per core) and 12 reducers (one per worker). An additional container is used to run the master. The 12 workers execute a WordCount benchmark on an implementation of MapReduce adapted to send the map results using DAIET. The input dataset is a 500 MB file containing random words that are not causing hash collisions.[5] We do not test with a larger dataset because the network aggregation is performed by bmv2 using mainly a single core. We configure P4 registers to store 16K key-value pairs, so that, with words of maximum 16 characters and a 4 B integer value, the total SRAM required would be around 10 MB, which is a reasonable amount of memory for a hardware P4 switch. To quantify the reduction, we run the same benchmark in two other baseline scenarios without in-network aggregation: (*i*) using the original TCP-based data exchange and (*ii*) using UDP and the DAIET protocol, but without executing data aggregation in the switch.

Figure 6.3 shows a box plot of the reduction, across all the workers, in the total data volume and execution time at the reducers observed with DAIET compared to the first baseline. We observe that in-network aggregation provides a 86.9%-89.3% reduction of the amount of data received by the reducers. Because the smaller the

[5] Our current prototype does not manage collisions.

data the less processing time at the reducer, we measured a median decrease of 83.6% in the execution time at the reducer, despite the received data are not sorted and require a complete sort operation.

Because current P4 hardware switches are expected to parse only around 200-300 B of each packet[6], we consider that one DAIET packet can contain at most 10 key-value pairs. Thus, our implementation generates more packets compared to the TCP baseline. However, the data volume reduction due to in-network aggregation is greater than the overhead caused by the additional packets. Figure 6.3 presents the reduction in the number of packets received by the reducer compared to the two baselines. We observe a median and maximum reduction of 90.5%, with a minimum of 88.1% compared with the baseline using UDP without in-network aggregation. Even considering the TCP baseline, we still measured a median 42% reduction in the number of packets. It is worth noting that an additional overhead in the data volume and number of packets is given by the fixed-size length of strings in our implementation, that forces a 16 B key even for smaller strings. This limitation will be removed in a future version of DAIET.

6.7 Related Work

Data Plane Applications. Emerging network devices offering data plane programmability paved the way for a whole new set of services that can be provided by these devices to support data center applications [100, 101] and improve network monitoring [102–105]. As an example, NetPaxos [107, 100] proposes a solution to implement the Paxos consensus protocol using programmable network devices. This is a valuable service that can help distributed data center applications to share a common state. In [135] programmable switches are used to implement approximate versions of several popular algorithms for network allocation. DC.p4 [136] shows how specific data-center switch features can be implemented with P4. SwitchKV [101] is a scalable key-value store, which enables content-based routing and efficient dynamic load balancing leveraging the switch hardware to match keys and forward the traffic to the right node at line rate. As these works showed, the programmable data plane can even execute fairly complicated algorithms, therefore we advocate

[6] According to private conversations with a P4 hardware vendor.

that the network can be used to execute also computation that, in the past, was only reserved to servers.

In-Network Aggregation. Several research efforts [124, 123] have been devoted for providing in-network aggregation for a variety of applications. NetAgg [124] is a software middlebox platform that provides an on-path aggregation service. NetAgg middleboxes are deployed on servers directly attached to network switches through high-bandwidth links, composing an aggregation tree in the network. It uses shim layers at edge servers to transparently intercept application traffic and redirect it to aggregation nodes. This requires changes in the network architecture which can be infeasible for already deployed data centers. Furthermore, for computation-bound applications the middleboxes can become a performance bottleneck. SHArP [123] is designed to offload MPI collective operation processing to the network. Reduction operations are performed on the data as it traverses a reduction tree in the network, reducing the volume of data as it goes up the tree. The nodes of the reduction tree are SHArP network devices that collect data from all their children and perform the aggregation operation once all the expected data is available. The results of the aggregation are distributed from the root of the reduction tree to the leaf nodes (i.e., the hosts), down the tree, or to a target InfiniBand Multi-cast group. SHArP suffers from a set of limitations: (*i*) it works only for MPI-based applications and does not support MapReduce, BSP or ML applications, (*ii*) it supports only a limited set of combiners, since they are directly implemented in the switch ASIC, and (*iii*) it always assumes that the aggregation root is the ToR switch, missing reductions opportunities that can be made at different levels of the network.

Unlike NetAgg and SHArP, DAIET does not modify the network architecture and provides more flexibility to support a variety of applications, including MapReduce, BSP, MPI and Machine Learning. Moreover, data plane programmability provides a greater flexibility, since support for new combiner functions can easily be added to the network. Similar to NetAgg, Camdoop [122] also supports on-path aggregation for MapReduce-based applications. It leverages the capabilities of Camcube [137] which uses direct-connect protocols where all traffic is forwarded between servers without switches. Thus, it requires a custom topology and it is incompatible with the common tree-based data center network infrastructure.

Besides data aggregation, IncBricks [108] is an in-network caching fabric with basic computing primitives. It leverages programmable switches and smart NICs.

It uses a key-value store as the application interface and allows to offload common compute operations on key-value pairs; e.g., increment, compare and update. Their design shifts the computation towards smart NICs since switches have limited storage. A specialized, in-switch key-value store for network measurement collection and aggregation also appears in Marple [106].

6.8 Conclusions

Programmable network hardware is finally emerging and provides the opportunity to revisit the idea of performing computation inside the network. Given ever more stringent requirements for data center applications facing hardware scalability bottlenecks and the end of Moore's law, programmable hardware appears to be the next frontier for achieving higher levels of efficiency and speed. Google's Tensor processing unit and Microsoft's Catapult projects are just two examples of this ongoing trend. We believe that the time has come to entrust network devices with part of the tasks typically executed by software. However, programmable networking devices have a distinct network machine architecture with stringent constraints. Determining the kinds of in-network computation, streaming algorithms and workloads that are going to be feasible under these architectural model is a major open challenge. As in the case of TCP offloading [138], we might need to see a period where variants are proposed, tested, evolved, and sometimes discarded. Data aggregation appears as a natural fit for in-network computation and our results are promising. But we view our work merely as an initial step towards the larger goal of judicious in-network computing.

We presented an early prototype of DAIET, a system that performs on-path aggregation for network-bound partition/aggregate data center applications. A preliminary evaluation shows that DAIET results in promising performance improvements for such applications, with the potential to significantly reduce job completion times. Differently from other approaches [124, 123], DAIET requires moderate modifications in the application software to support the communication protocol and deal with unordered intermediate results. Moreover, while the correctness of the final results is preserved, there is no guarantee that all the intermediate results are aggregated. Thus, the final worker still has to perform an aggregation function, albeit on a smaller amount of data.

While our current approach has several limitations, its prototype implementation allowed us to evaluate DAIET and determine the main challenges to address. As future work, we plan to extend this approach in multiple ways. First, reliability must be addressed, defining a protocol to identify and recover packet losses. Secondly, we intend to extend DAIET to support a wide range of partition-aggregate applications, such as deep learning, graph and stream processing. Finally, we plan to extensively evaluate DAIET on these applications using hardware programmable devices.

Chapter 7

Conclusions

This dissertation shows how the different components of a modern NSP infrastructure can be used to provide several services designed factoring in their different characteristics and constraints. Some of these characteristics (e.g., being traversed by network traffic) favor a class of services that can be deployed locally, while the specific limitations, especially in term of computational power, require a judicious design of the service architecture.

We show different novel solutions to distribute services on different parts of the heterogeneous NSP infrastructure. In chapter 3 we introduce the concept of *Native Network Functions*, software components that can be executed on CPEs with low hardware resources taking advantage of the proximity to the user to provide low latency services. More complex functionalities can be executed remotely in a data center in the form of VNFs. An overarching orchestrator manages the lifecycle of both NNFs and VNFs, possibly moving them from the CPE to the data center (or viceversa) at runtime when the live traffic and expected performance require an higher computational power (or lower latency).

An estimate of the performance of NFs on the different available platforms is of paramount importance to the decision-making process of the orchestrator, especially for the choice of the best placement plan that would reduce the possibility of migration of services at runtime. We present in chapter 2 a novel approach to NF modeling that takes into account the functionality of the NF at hand. Moreover, we show how this model can be mapped on different platforms to get performance estimates that take into account the different hardware features.

In chapter 4 we show *U-Filter*, an instance of distributed service that exploits both the proximity to the user of the CPE and the higher computation capabilities of the data center to perform URL filtering with low impact on the user experience. Considering the limitations of common CPEs, we present an efficient solution that performs a limited form of DPI in the kernel of the residential gateway OS and delegates the complex policy checking process to a remote server. This server is in charge of keeping the large URL database up-to-date and can serve multiple CPEs. The operation of *U-Filter* is carefully designed to reduce the impact of the policy checking on the web page loading speed. Our experiments showed that the browsing experience is unaltered in a typical deployment scenario.

Chapter 5 focuses on the access and core network, typically solely dedicated to traffic forwarding. Our proposal shows that, leveraging the favorable position in the path of the traffic and an intelligent traffic assignment algorithm, a network of nodes can also perform packet processing to extract additional data from the traffic itself. The straighforward application is traffic inspection and monitoring, but our approach is general enough that can be used for different scenarios where data flowing through a network of nodes require processing, such as the analysis of sensors data in an IoT environment.

Finally in chapter 6 we consider the data center network fabric and the recent trend of programmable data plane hardware. We want to encourage the community to think of new solutions to offload part of the computation on the modern programmable networking hardware and to co-design data center distributed systems with their network layer. We show one proof-of-concept that shows the feasibility and the benefit of this approach. Our solution performs an in-network aggregation service that is useful to various typical data center workloads. However, our work is just an initial step in the direction of providing in-network services in the data center.

In conclusion, this thesis shows that, with modern networks, many opportunities arise for deploying services throughout the network infrastructure. NFV and programmable dataplane are two of the key enabler to provide additional distributed services that can ease network management, reduce network overhead and be a new source of revenues for service providers. However, the architecture of these services must be careful designed and new algorithms are needed to match the diverse challenges that in-network computing poses.

References

[1] AT&T DirectTV. https://www.att.com/bundles/data-free-tv.html. [Online; accessed 14-Feb-2018].

[2] Orange TV. https://boutique.orange.fr/tv. [Online; accessed 14-Feb-2018].

[3] TIM Vision. https://www.tim.it/smart-life/tv-entertainment/tv/timvision. [Online; accessed 14-Feb-2018].

[4] Vodafone Rete Sicura. http://www.vodafone.it/portal/Privati/Vantaggi-Vodafone/La-nostra-Rete-Veloce/Rete-Sicura/Vodafone-Rete-Sicura. [Online; accessed 14-Feb-2018].

[5] Binge On. https://www.t-mobile.com/offer/binge-on-streaming-video.html. [Online; accessed 14-Feb-2018].

[6] Comcast Stream TV. https://www.theverge.com/2015/11/21/9776052/comcast-stream-tv-data-cap-exemption-net-neutrality. [Online; accessed 14-Feb-2018].

[7] Mario Baldi and Amedeo Sapio. A network function modeling approach for performance estimation. In *1st International Forum on Research and Technologies for Society and Industry (RTSI)*. IEEE, 2015.

[8] Amedeo Sapio, Mario Baldi, and Gergely Pongrácz. Cross-Platform Estimation of Network Function Performance. In *EWSDN*. IEEE, 2015.

[9] Mario Baldi and Amedeo Sapio. Network Function Modeling and Performance Estimation. *International Journal of Electrical and Computer Engineering*, 2018.

[10] ETSI ISG for NFV, ETSI GS NFV-INF 001, Network Functions Virtualisation (NFV); Infrastructure Overview. http://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/001/01.01.01_60/gs_NFV-INF001v010101p.pdf. [Online; accessed 18-February-2017].

[11] Mario Baldi, Roberto Bonafiglia, Fulvio Risso, and Amedeo Sapio. Modeling Native Software Components as Virtual Network Functions. In *SIGCOMM Conference*. ACM, 2016.

[12] Anil Rijsinghani. RFC 1624 - Computation of the Internet Checksum via Incremental Update. https://tools.ietf.org/html/rfc1624, 1994. [Online; accessed 19-December-2016].

[13] Gergely Pongrácz, László Molnár, Zoltán Lajos Kis, and Zoltán Turányi. Cheap silicon: a myth or reality? picking the right data plane hardware for software defined networking. In *HotSDN*. ACM, 2013.

[14] Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html. [Online; accessed 18-February-2017].

[15] Intel DPDK: Data Plane Development Kit. http://dpdk.org. [Online; accessed 19-March-2017].

[16] Intel Ivy Bridge Benchmark. http://www.7-cpu.com/cpu/IvyBridge.html. [Online; accessed 18-February-2017].

[17] Pankaj Gupta, Steven Lin, and Nick McKeown. Routing lookups in hardware at memory access speeds. In *INFOCOM*. IEEE, 1998.

[18] Intel Data Plane Performance Demonstrators. https://01.org/intel-data-plane-performance-demonstrators. [Online; accessed 18-February-2017].

[19] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The Design and Implementation of Open vSwitch. In *NSDI*. USENIX, 2015.

[20] Ntop PF_RING. http://www.ntop.org/products/packet-capture/pf_ring/. [Online; accessed 19-March-2017].

[21] Open vSwitch Manual - ovs-ofctl. http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt. [Online; accessed 19-March-2017].

[22] Dilip Joseph and Ion Stoica. Modeling middleboxes. *IEEE Network*, 2008.

[23] Aaron Gember, Anand Krishnamurthy, Saul St John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. *arXiv preprint arXiv:1305.0209*, 2013.

[24] Joao Soares, Miguel Dias, Jorge Carapinha, Bruno Parreira, and Susana Sargento. Cloud4NFV: A platform for Virtual Network Functions. In *CloudNet*. IEEE, 2014.

[25] Francesco Lucrezia, Guido Marchetto, Fulvio Giovanni Ottavio Risso, and Vinicio Vercellone. Introducing Network-Aware Scheduling Capabilities in OpenStack. In *NetSoft*. IEEE, 2015.

[26] Nikolaus Huber, Marcel von Quast, Michael Hauck, and Samuel Kounev. Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments. In *CLOSER*, 2011.

[27] Ravi Iyer, Ramesh Illikkal, Omesh Tickoo, Li Zhao, Padma Apparao, and Don Newell. VM 3: Measuring, modeling and managing VM shared resources. *Computer Networks*, 2009.

[28] Timothy Wood, Ludmila Cherkasova, Kivanc Ozonat, and Prashant Shenoy. Profiling and modeling resource usage of virtualized applications. In *International Conference on Middleware*. Springer-Verlag, 2008.

[29] Roberto Bonafiglia, Sebastiano Miano, Sergio Nuccio, Fulvio Risso, and Amedeo Sapio. Enabling NFV services on resource-constrained CPEs. In *CloudNet*. IEEE, 2016.

[30] András Császár, Wolfgang John, Mario Kind, Catalin Meirosu, Gergely Pongrácz, Dimitri Staessens, Attila Takács, and Fritz-Joachim Westphal. Unifying Cloud and Carrier Network: EU FP7 Project UNIFY. In *UCC*. IEEE, 2013.

[31] Zvika Bronstein and Eyal Shraga. NFV virtualisation of the home environment. In *CCNC*. IEEE, 2014.

[32] Tiago Cruz, Paulo Simões, Nuno Reis, Edmundo Monteiro, Fernando Bastos, and Alexandre Laranjeira. An architecture for virtualized home gateways. In *IM*. IEEE, 2013.

[33] Nicolas Herbaut, Daniel Negru, George Xilouris, and Yiping Chen. Migrating to a nfv-based home gateway: introducing a surrogate vnf approach. In *NOF*. IEEE, 2015.

[34] Fernando Sanchez and David Brazewell. Tethered Linux CPE for IP service delivery. In *NetSoft*. IEEE, 2015.

[35] Giuseppe Faraci and Giovanni Schembra. An analytical model to design and manage a green SDN/NFV CPE node. *Transactions on Network and Service Management*, 2015.

[36] Ivano Cerrato, Alex Palesandro, Fulvio Risso, Marc Suñé, Vinicio Vercellone, and Hagen Woesner. Toward dynamic virtualized network services in telecom operator networks. *Computer Networks*, 2015.

[37] ETSI ISG for NFV, ETSI GS NFV-MAN 001, Network Functions Virtualisation (NFV) Management and Orchestration. http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf. [Online; accessed 7-March-2016].

[38] Tao Su, Antonio Lioy, and Nicola Barresi. Trusted computing technology and proposals for resolving cloud computing security problems. *Cloud Computing Security: Foundations and Challenges*, 2016.

[39] Strongswan. https://www.strongswan.org/. [Online; accessed 7-August-2016].

[40] Roberto Bonafiglia, Amedeo Sapio, Mario Baldi, Fulvio Risso, and Paolo C. Pomi. Enforcement of dynamic HTTP policies on resource-constrained residential gateways. *Computer Networks*, 2017.

[41] Google Fiber. https://fiber.google.com. [Online; accessed 17-March-2017].

[42] DansGuardian. http://dansguardian.org. [Online; accessed 17-March-2017].

[43] Mobile Fence - Parental Control. http://www.mobilefence.com. [Online; accessed 17-March-2017].

[44] Cloudacl WebFilter. http://www.cloudacl.com. [Online; accessed 17-March-2017].

[45] The netfilter.org project. http://netfilter.org/. [Online; accessed 17-March-2017].

[46] Matthew V. Mahoney. Network Traffic Anomaly Detection Based on Packet Bytes. In *Symposium on Applied Computing*. ACM, 2003.

[47] Sungkornsarun Longchupole, Noppadol Maneerat, and Ruttikorn Varakulsiripunth. Anomaly detection through packet header data. In *ICICS*. IEEE, 2009.

[48] Seong Soo Kim and A. L. Narasimha Reddy. Statistical Techniques for Detecting Traffic Anomalies Through Packet Header Data. *Transactions on Networking*, 2008.

[49] Andrew W. Moore and Konstantina Papagiannaki. Toward the accurate identification of network applications. In *PAM*. Springer, 2005.

[50] William Richard Stevens. *UNIX Network Programming: Networking APIs*, volume 1. Prentice-Hall, Inc., 1997.

[51] Fabian Schneider, Bernhard Ager, Gregor Maier, Anja Feldmann, and Steve Uhlig. Pitfalls in HTTP traffic measurements and analysis. In *PAM*. Springer, 2012.

[52] Bryce Thomas, Raja Jurdak, and Ian Atkinson. SPDYing up the web. *Communications of the ACM*, 2012.

[53] Liang Shuai, Gaogang Xie, and Jianhua Yang. Characterization of HTTP behavior on access networks in Web 2.0. In *ICT*. IEEE, 2008.

[54] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *SIGCOMM CCR*. ACM, 2015.

[55] Xingliang Yuan, Xinyu Wang, Jianxiong Lin, and Cong Wang. Privacy-preserving deep packet inspection in outsourced middleboxes. In *INFOCOM*. IEEE, 2016.

[56] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context tls (mctls): Enabling secure in-network functionality in tls. In *SIGCOMM CCR*. ACM, 2015.

[57] Salvatore Loreto, John Mattsson, Robert Skog, Hans Spaak, Gus Bourg, Dan Druta, and Mohammad Hafeez. Explicit trusted proxy in http/2.0. https://tools.ietf.org/html/draft-loreto-httpbis-trusted-proxy20-01, 2014. [Online; accessed 17-March-2017].

[58] Amedeo Sapio, Yong Liao, Mario Baldi, Gyan Ranjan, Fulvio Risso, Alok Tongaonkar, Ruben Torres, and Antonio Nucci. Per-user Policy Enforcement on Mobile Apps Through Network Functions Virtualization. In *MobiArch*. ACM, 2014.

[59] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. The cost of the s in https. In *CoNEXT*. ACM, 2014.

[60] OpenWrt: Linux distribution for embedded devices. https://openwrt.org. [Online; accessed 17-March-2017].

[61] ApacheBench. http://httpd.apache.org/docs/2.2/programs/ab.html. [Online; accessed 17-March-2017].

[62] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. RFC 2018 - TCP Selective Acknowledgment Options. https://www.ietf.org/rfc/rfc2018.txt, 1996. [Online; accessed 17-March-2017].

[63] Mark Allman, Vern Paxson, and Ethan Blanton. TCP Congestion Control. https://www.ietf.org/rfc/rfc5681.txt, 2009. [Online; accessed 17-March-2017].

[64] Marco Mellia, Renato Lo Cigno, and Fabio Neri. Measuring IP and TCP behavior on edge nodes with Tstat. *Computer Networks*, 2005.

[65] Tinyproxy. http://tinyproxy.github.io. [Online; accessed 17-March-2017].

[66] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. https://tools.ietf.org/html/rfc2616.txt, 1999. [Online; accessed 17-March-2017].

[67] K9 Web Protection. http://www1.k9webprotection.com. [Online; accessed 17-March-2017].

[68] Cisco Umbrella. https://umbrella.cisco.com/use-cases/web-filtering. [Online; accessed 17-March-2017].

[69] Blue Coat WebFilter. https://www.bluecoat.com/products/webfilter. [Online; accessed 17-March-2017].

[70] Li-Der Chou, Zheng He, David Chunhu Li, Hui-Fan Chen, Jun-Jie Su, Ching-Yung Chen, Hsu-Chuan Wei, and Chia-Jen Li. Design and implementation of content-based filter system on embedded linux home gateway. In *ICACT*. IEEE, 2012.

[71] Richard Cziva, Simon Jouet, and Dimitrios P Pezaros. Roaming edge vnfs using glasgow network functions. In *SIGCOMM Conference*. ACM, 2016.

[72] Amedeo Sapio, Mario Baldi, Fulvio Risso, Narendra Anand, and Antonio Nucci. Packet Capture and Analysis on MEDINA, A Massively Distributed Network Data Caching Platform. *Parallel Processing Letters*, 2017.

[73] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying Data Deduplication. In *Middleware Conference Companion*. ACM/IFIP/USENIX, 2008.

[74] Dutch T. Meyer and William J. Bolosky. A Study of Practical Deduplication. In *FAST*. USENIX, 2011.

[75] Cisco UCS E-Series Servers. http://www.cisco.com/c/en/us/products/servers-unified-computing/ucs-e-series-servers/index.html. [Online; accessed 27-January-2017].

[76] Chia-Wei Chang, Guanyao Huang, Bill Lin, and Chen-Nee Chuah. Leisure: Load-balanced network-wide traffic measurement and monitor placement. *Transactions on Parallel and Distributed Systems*, 2015.

[77] Ruediger Gad, Martin Kappes, and Inmaculada Medina-Bulo. Monitoring traffic in computer networks with dynamic distributed remote packet capturing. In *ICC*. IEEE, 2015.

[78] Luis M. Vaquero and Luis Rodero-Merino. Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. *SIGCOMM CCR*, 2014.

[79] Alexandr Krylovskiy, Marco Jahn, and Edoardo Patti. Designing a smart city internet of things platform with microservice architecture. In *FiCloud*. IEEE, 2015.

[80] Ruediger Gad, Martin Kappes, Robin Mueller-Bady, and Inmaculada Medina-Bulo. Header field based partitioning of network traffic for distributed packet capturing and processing. In *AINA*. IEEE, 2014.

[81] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. CSAMP: A System for Network-wide Flow Monitoring. In *NSDI*. USENIX, 2008.

[82] Adam Kirsch, Michael Mitzenmacher, and George Varghese. Hash-based techniques for high-speed packet processing. In *Algorithms for Next Generation Networks*. Springer, 2010.

[83] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 1987.

[84] Colleen Shannon, David Moore, and K. C. Claffy. Beyond Folklore: Observations on Fragmented Traffic. *Transactions on Networking*, 2002.

[85] Bing Xiong, Kun Yang, Feng Li, Xiaosu Chen, Jianming Zhang, Qiang Tang, and Yuansheng Luo. The impact of bitwise operators on hash uniformity in network packet processing. *International Journal of Communication Systems*, 2014.

[86] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational IP networks: Methodology and experience. *Transactions on Networking*, 2001.

[87] Marco Chiesa, Christoph Dietzel, Gianni Antichi, Marc Bruyere, Ignacio Castro, Mitch Gusat, Thomas King, Andrew W Moore, Thanh Dang Nguyen, Philippe Owezarski, et al. Inter-domain networking innovation on steroids: empowering ixps with SDN capabilities. *IEEE Communications Magazine*, 2016.

[88] Matevz Pustisek, Iztok Humar, and Janez Bester. Empirical analysis and modeling of peer-to-peer traffic flows. In *MELECON*. IEEE, 2008.

[89] Manish R Sharma and John W Byers. Scalable coordination techniques for distributed network monitoring. In *PAM*. Springer, 2005.

[90] Arpit Gupta, Rüdiger Birkner, Marco Canini, Nick Feamster, Chris Mac-Stoker, and Walter Willinger. Network Monitoring As a Streaming Analytics Problem. In *HotNets*. ACM, 2016.

[91] Vyas Sekar, Anupam Gupta, Michael K Reiter, and Hui Zhang. Coordinated sampling sans origin-destination identifiers: algorithms and analysis. In *COMSNETS*. IEEE, 2010.

[92] Ashok Anand, Vyas Sekar, and Aditya Akella. SmartRE: an architecture for coordinated network-wide redundancy elimination. In *SIGCOMM CCR*. ACM, 2009.

[93] Andrea Di Pietro, Felipe Huici, Diego Costantini, and Saverio Niccolini. Decon: Decentralized coordination for large-scale flow monitoring. In *INFOCOM*. IEEE, 2010.

[94] Shan-Hsiang Shen and Aditya Akella. Decor: a distributed coordinated resource monitoring system. In *IWQoS*. IEEE, 2012.

[95] Noriaki Kamiyama, Tatsuya Mori, and Ryoichi Kawahara. Autonomic load balancing of flow monitors. *Computer Networks*, 2013.

[96] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *HotNets*. ACM, 2017.

[97] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM Conference*. ACM, 2013.

[98] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM CCR*, 2014.

[99] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM Conference*. ACM, 2016.

[100] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos Made Switch-y. *SIGCOMM CCR*, 2016.

[101] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. Be Fast, Cheap and in Control with SwitchKV. In *NSDI*. USENIX, 2016.

[102] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data Plane Performance Diagnosis of TCP. In *SOSR*. ACM, 2017.

[103] Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. Heavy-Hitter Detection Entirely in the Data Plane. In *SOSR*. ACM, 2017.

[104] Diana Andreea Popescu, Gianni Antichi, and Andrew W Moore. Enabling Fast Hierarchical Heavy Hitter Detection using Programmable Data Planes. In *SOSR*. ACM, 2017.

[105] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band Network Telemetry via Programmable Dataplanes. In *SOSR*. ACM, 2015.

[106] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *SIGCOMM Conference*. ACM, 2017.

[107] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *SOSR*. ACM, 2015.

[108] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *ASPLOS*. ACM, 2017.

[109] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*. ACM, 2017.

[110] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. *SIGCOMM CCR*, 1996.

[111] Barefoot Networks. Barefoot Tofino. "https://barefootnetworks.com/products/brief-tofino/". [Online; accessed 14-Feb-2018].

[112] Cavium. XPliant Ethernet Switch Product Family. "http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html". [Online; accessed 14-Feb-2018].

[113] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *Transactions on Computer Systems*, 1984.

[114] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*. USENIX, 2004.

[115] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*. USENIX, 2014.

[116] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[117] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. USENIX, 2016.

[118] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD*. ACM, 2010.

[119] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*. USENIX, 2012.

[120] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *NSDI*. USENIX, 2016.

[121] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*. USENIX, 2008.

[122] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In *NSDI*. USENIX, 2012.

[123] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *COM-HPC*, 2016.

[124] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L Wolf. NetAgg: Using Middleboxes for Application-Specific On-path Aggregation in Data Centres. In *CoNEXT*. ACM, 2014.

[125] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@Twitter. In *SIGMOD*. ACM, 2014.

[126] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM CCR*. ACM, 2009.

[127] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast congestion control for TCP in data-center networks. *Transactions on Networking*, 2013.

[128] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling Packet Programs to Reconfigurable Switches. In *NSDI*. USENIX, 2015.

[129] Netronome. Agilio CX SmartNICs. "https://www.netronome.com/products/agilio-cx/". [Online; accessed 27-Mar-2017].

[130] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM CCR*, 2008.

[131] Barefoot Networks. The World's Fastest & Most Programmable Networks. "https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/". [Online; accessed 14-Feb-2018].

[132] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, 2014.

[133] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *SSDBM*, 2013.

[134] Amedeo Sapio, Ibrahim Abdelaziz, Marco Canini, and Panos Kalnis. DAIET: a system for data aggregation inside the network. In *SoCC*. ACM, 2017.

[135] Naveen Kr Sharma, Antoine Kaufmann, Thomas Anderson, Changhoon Kim, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *NSDI*. USENIX, 2017.

[136] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. Dc.p4: Programming the forwarding plane of a data-center switch. In *SOSR*. ACM, 2015.

[137] Hussam Abu-Libdeh, Paolo Costa, Antony Rowstron, Greg O'Shea, and Austin Donnelly. Symbiotic Routing in Future Data Centers. In *SIGCOMM Conference*. ACM, 2010.

[138] Jeffrey C Mogul. TCP Offload Is a Dumb Idea Whose Time Has Come. In *HotOS*. ACM, 2003.